Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

A Formal C Memory Model for Separation Logic

Robbert Krebbers

Received: n/a

Abstract The core of a formal semantics of an imperative programming language is a memory model that describes the behavior of operations on the memory. Defining a memory model that matches the description of C in the C11 standard is challenging because C allows both *high-level* (by means of typed expressions) and *low-level* (by means of bit manipulation) memory accesses. The C11 standard has restricted the interaction between these two levels to make more effective compiler optimizations possible, at the expense of making the memory model complicated.

We describe a formal memory model of the (non-concurrent part of the) C11 standard that incorporates these restrictions, and at the same time describes low-level memory operations. This formal memory model includes a rich permission model to make it usable in separation logic and supports reasoning about program transformations. The memory model and essential properties of it have been fully formalized using the Coq proof assistant.

Keywords ISO C11 Standard · C Verification · Memory Models · Separation Logic · Interactive Theorem Proving · Coq

1 Introduction

A memory model is the core of a semantics of an imperative programming language. It models the memory states and describes the behavior of memory operations. The main operations described by a C memory model are:

- Reading a value at a given address.
- Storing a value at a given address.
- Allocating a new object to hold a local variable or storage obtained via `malloc`.
- Deallocating a previously allocated object.

Robbert Krebbers
ICIS, Radboud University Nijmegen, The Netherlands
Aarhus University, Denmark
E-mail: mail@robbertkrebbers.nl

Formalizing the C11 memory model in a faithful way is challenging because C features both *low-level* and *high-level* data access. Low-level data access involves unstructured and untyped byte representations whereas high-level data access involves typed abstract values such as arrays, structs and unions.

This duality makes the memory model of C more complicated than the memory model of nearly any other programming language. For example, more mathematically oriented languages such as Java and ML feature only high-level data access, in which case the memory can be modeled in a relatively simple and structured way, whereas assembly languages feature only low-level data access, in which case the memory can be modeled as an array of bits.

The situation becomes more complicated as the C11 standard allows compilers to perform optimizations based on a high-level view of data access that are inconsistent with the traditional low-level view of data access. This complication has lead to numerous ambiguities in the standard text related to aliasing, uninitialized memory, end-of-array pointers and type-punning that cause problems for C code when compiled with widely used compilers. See for example the message [42] on the standard committee's mailing list, Defect Reports #236, #260, and #451 [26], and the various examples in this paper.

Contribution. This paper describes the CH_2O memory model, which is part of the the CH_2O project [30–37]. CH_2O provides an operational, executable and axiomatic semantics in Coq for a large part of the non-concurrent fragment of C, based on the official description of C as given by the C11 standard [27].

The key features of the CH_2O memory model are as follows:

- **Close to C11.** CH_2O is faithful to the C11 standard in order to be compiler independent. When one proves something about a given program with respect to CH_2O, it should behave that way with *any* C11 compliant compiler (possibly restricted to certain implementation-defined choices).
- **Static type system.** Given that C is a statically typed language, CH_2O does not only capture the dynamic semantics of C11 but also its type system. We have established properties such as type preservation of the memory operations.
- **Proof infrastructure.** All parts of the CH_2O memory model and semantics have been formalized in Coq (without axioms). This is essential for its application to program verification in proof assistants. Also, considering the significant size of CH_2O and its memory model, proving metatheoretical properties of the language would have been intractable without the support of a proof assistant.
  Despite our choice to use Coq, we believe that nearly all parts of CH_2O could be formalized in any proof assistant based on higher-order logic.
- **Executable.** To obtain more confidence in the accuracy of CH_2O with respect to C11, the CH_2O memory model is executable. An executable memory model allows us to test the CH_2O semantics on example programs and to compare the behavior with that of widely used compilers [33, 37].
- **Separation logic.** In order to reason about concrete C programs, one needs a program logic. To that end, the CH_2O memory model incorporates a complex permission model suitable for separation logic. This permission system, as well as the memory model itself, forms a separation algebra.
- **Memory refinements.** CH_2O has an expressive notion of memory refinements that relates memory states. All memory operations are proven invariant under

this notion. Memory refinements form a general way to validate many commonsense properties of the memory model in a formal way. They also open the door to reasoning about program transformations, which is useful if one were to use the memory model as part of a verified compiler front-end.

This paper is an extended version of previously published conference papers at CPP [30] and VSTTE [32]. In the time following these two publications, the memory model has been extended significantly and been integrated into an operational, executable and axiomatic semantics [33, 37]. The memory model now supports more features, various improvements to the definitions have been carried out, and more properties have been formally proven as part of the Coq development.

Parts of this paper also appear in the author's PhD thesis [33], which describes the entire CH_2O project including its operational, executable and axiomatic semantics, and metatheoretical results about these.

Problem. The C11 standard gives compilers a lot of freedom in what behaviors a program may have [27, 3.4]. It uses the following notions of under-specification:

- *Unspecified behavior*: two or more behaviors are allowed. For example: the execution order of expressions. The choice may vary for each use of the construct.
- *Implementation-defined behavior*: like unspecified behavior, but the compiler has to document its choice. For example: size and endianness of integers.
- *Undefined behavior:* the standard imposes no requirements at all, the program is even allowed to crash. For example: dereferencing a `NULL` pointer, or signed integer overflow.

Under-specification is used extensively to make C portable, and to allow compilers to generate fast code. For example, when dereferencing a pointer, no code has to be generated to check whether the pointer is valid or not. If the pointer is invalid (`NULL` or a dangling pointer), the compiled program may do something arbitrary instead of having to exit with a `NullPointerException` as in Java. Since the CH_2O semantics intends to be a formal version of the C11 standard, it has to capture the behavior of *any* C compiler, and thus has to take *all* under-specification seriously (even if that makes the semantics complex).

Modeling under-specification in a formal semantics is folklore: unspecified behavior corresponds to non-determinism, implementation-defined behavior corresponds to parameterization, and undefined behavior corresponds to a program having no semantics. However, the extensive amount of underspecification in the C11 standard [27, Annex J], and especially that with respect to the memory model, makes the situation challenging. We will give a motivating example of subtle underspecification in the introduction of this paper. Section 3 provides a more extensive overview.

Motivating example. A drawback for efficient compilation of programming languages with pointers is *aliasing*. Aliasing describes a situation in which multiple pointers refer to the same object. In the following example the pointers `p` and `q` are said to be *aliased*.

```
int x; int *p = &x, *q = &x;
```

The problem of aliased pointers is that writes through one pointer may effect the result of reading through the other pointer. The presence of aliased pointers therefore often disallows one to change the order of instructions. For example, consider:

```
int f(int *p, int *q) {
  int z = *q; *p = 10; return z;
}
```

When `f` is called with pointers `p` and `q` that are aliased, the assignment to `*p` also affects `*q`. As a result, one cannot transform the function body of `f` into the shorter `*p = 10; return (*q);`. The shorter function will return 10 in case `p` and `q` are aliased, whereas the original `f` will always return the original value of `*q`.

Unlike this example, there are many situations in which pointers can be assumed *not* to alias. It is essential for an optimizing compiler to determine where aliasing cannot occur, and use this information to generate faster code. The technique of determining whether pointers can alias or not is called *alias analysis*.

In *type-based alias analysis*, type information is used to determine whether pointers can alias or not. Consider the following example:

```
short g(int *p, short *q) {
  short z = *q; *p = 10; return z;
}
```

Here, a compiler is allowed to assume that `p` and `q` are not aliased because they point to objects of different types. The compiler is therefore allowed to transform the function body of `g` into the shorter `*p = 10; return (*q);`.

The peculiar thing is that the C type system does not statically enforce the property that pointers to objects of different types are not aliased. A union type can be used to create aliased pointers to different types:

```
union int_or_short { int x; short y; } u = { .y = 3 };
int *p = &u.x;   // p points to the x variant of u
short *q = &u.y; // q points to the y variant of u
return g(p, q);  // g is called with aliased pointers p and q
```

The above program is valid according to the rules of the C11 type system, but has undefined behavior during execution of `g`. This is caused by the standard's notion of *effective types* [27, 6.5p6-7] (also called *strict-aliasing restrictions*) that assigns undefined behavior to incorrect usage of aliased pointers to different types.

We will inline part of the function body of `g` to indicate the incorrect usage of aliased pointers during the execution of the example.

```
union int_or_short { int x; short y; } u = { .y = 3 };
int *p = &u.x;   // p points to the x variant of u
short *q = &u.y; // q points to the y variant of u
// g(p, q) is called, the body is inlined
short z = *q;    // u has variant y and is accessed through y -> OK
*p = 10;         // u has variant y and is accessed through x -> bad
```

The assignment `*p = 10` violates the rules for effective types. The memory area where `p` points to contains a union whose variant is `y` of type **`short`**, but is accessed through a pointer to variant `x` of type **`int`**. This causes undefined behavior.

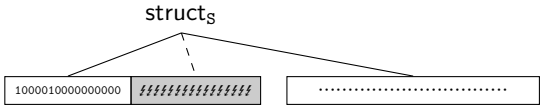
Effective types form a clear tension between the low-level and high-level way of data access in C. The low-level representation of the memory is inherently untyped and unstructured and therefore does not contain any information about variants of unions. However, the standard treats the memory as if it were typed.

Approach. Most existing C formalizations (most notably Norrish [45], Leroy *et al.* [39, 40] and Ellison and Roşu [19]) use an unstructured untyped memory model where each object in the formal memory model consists of an array of bytes. These formalizations therefore cannot assign undefined behavior to violations of the rules for effective types, among other things.

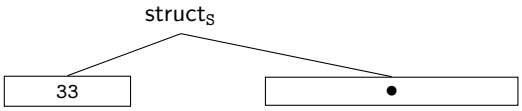
In order to formalize the interaction between low-level and high-level data access, and in particular effective types, we represent the formal memory state as a forest of well-typed trees whose structure corresponds to the structure of data types in C. The leaves of these trees consist of bits to capture low-level aspects of the language.

The key concepts of our memory model are as follows:

- *Memory trees* (Section 6.3) are used to represent each object in memory. They are abstract trees whose structure corresponds to the shape of C data types. The memory tree of `struct S { short x, *r; } s = { 33, &s.x }` might be (the precise shape and the bit representations are implementation defined):



  The leaves of memory trees contain permission annotated bits (Section 6.2). Bits are represented symbolically: the integer value `33` is represented as its binary representation `1000010000000000`, the padding bytes as symbolic *indeterminate* bits ↯ (whose actual value should not be used), and the pointer `&s.x` as a sequence of symbolic *pointer bits*.
  The *memory* itself is a forest of memory trees. Memory trees are explicit about type information (in particular the variants of unions) and thus give rise to a natural formalization of effective types.
- *Pointers* (Section 6.1) are formalized using paths through memory trees. Since we represent pointers as paths, the formal representation contains detailed information about how each pointer has been obtained (in particular which variants of unions were used). A detailed formal representation of pointers is essential to describe effective types.
- *Abstract values* (Definition 6.4) are trees whose structure is similar to memory trees, but have base values (mathematical integers and pointers) on their leaves. The abstract value of `struct S { short x, *r; } s = { 33, &s.x }` is:



  Abstract values hide internal details of the memory such as permissions, padding and object representations. They are therefore used in the external interface of the memory model and throughout the operational semantics.

Memory trees, abstract values and bits with permissions can be converted into each other. These conversions are used to define operations internal to the memory model. However, none of these conversions are bijective because different information is materialized in these three data types:

	Abstract values	Memory trees	Bits with permissions
Permissions		✓	✓
Padding		always ↯	✓
Variants of union	✓	✓	
Mathematical values	✓		

This table indicates that abstract values and sequences of bits are complementary. Memory trees are a middle ground, and therefore suitable to describe both the low-level and high-level aspects of the C memory.

Outline. This work presents an executable mathematically precise version of a large part of the (non-concurrent) C memory model.

- Section 3 describes some challenges that a C11 memory model should address; these include end-of-array pointers, byte-level operations, indeterminate memory, and type-punning.
- Section 4 describes the types of C. Our formal development is parameterized by an abstract interface to characterize implementation-defined behavior.
- Section 5 describes the permission model using a variant of separation algebras that is suitable for formalization in Coq. The permission model is built compositionally from simple separation algebras.
- Section 6 contains the main part of this paper, it describes a memory model that can accurately deal with the challenges posed in Section 3.
- Section 7 demonstrates that our memory model is suitable for formal proofs. We prove that the standard's notion of effective types has the desired effect of allowing type-based alias analysis (Section 7.1), we present a method to reason compositionally about memory transformations (Section 7.2), and prove that the memory model has a separation algebra structure (Section 7.4).
- Section 8 describes the Coq formalization: all proofs about our memory model have been fully formalized using Coq.

As this paper describes a large formalization effort, we often just give representative parts of definitions. The interested reader can find all details online as part of our Coq formalization at:

`http://robbertkrebbers.nl/research/ch2o/`.

2 Notations

This section introduces some common mathematical notions and notations that will be used throughout this paper.

Definition 2.1 We let $\mathbb{N}$ denote the type of *natural numbers* (including 0), let $\mathbb{Z}$ denote the type of *integers*, and let $\mathbb{Q}$ denote the type of *rational numbers*. We let $i \mid j$ denote that $i \in \mathbb{N}$ is a *divisor* of $j \in \mathbb{N}$.

Definition 2.2 We let Prop denote the type of *propositions*, and let bool denote the type of *Booleans* whose elements are true and false. Most propositions we consider have a corresponding Boolean-valued decision function. In Coq we use type classes to keep track of these correspondences, but in this paper we leave these correspondences implicit.

Definition 2.3 We let $\mathsf{option}\ A$ denote the *option type over* A, whose elements are inductively defined as either $\bot$ or x for some $x \in A$. We implicitly lift operations to operate on the option type, and often omit cases of definitions that yield $\bot$. This is formally described using the *option monad* in the Coq formalization.

Definition 2.4 A *partial function* f *from* A *to* B is a function $f : A \to \mathsf{option}\ B$.

Definition 2.5 A partial function f is called *a finite partial function* or a *finite map* if its *domain* $\mathsf{dom}\ f := \{x \mid \exists y \in B\,.\, f\,x = y\}$ is finite. The type of finite partial functions is denoted as $A \to_{\mathsf{fin}} B$. The operation $f[x := y]$ yields f with the value y for argument x.

Definition 2.6 We let $A \times B$ denote *the product of types* A *and* B. Given a *pair* $(x, y) \in A \times B$, we let $(x, y)_{\mathbf{1}} := x$ and $(x, y)_{\mathbf{2}} := y$ denote the *first* and *second projection* of (x, y).

Definition 2.7 We let $\mathsf{list}\ A$ denote the *list type over* A, whose elements are inductively defined as either ε or $x\,\vec{x}$ for some $x \in A$ and $\vec{x} \in \mathsf{list}\ A$. We let $x_i \in A$ denote the ith element of a list $\vec{x} \in \mathsf{list}\ A$ (we count from 0). Lists are sometimes denoted as $[\,x_0, \ldots, x_{n-1}\,] \in \mathsf{list}\ A$ for $x_0, \ldots, x_{n-1} \in A$.

We use the following operations on lists:

- We often implicitly lift a function $f : A_0 \to \cdots \to A_n$ point-wise to the function $f : \mathsf{list}\ A_0 \to \cdots \to \mathsf{list}\ A_n$. The resulting list is truncated to the length of the smallest input list in case $n > 1$.
- We often implicitly lift a predicate $P : A_0 \to A_{n-1} \to \mathsf{Prop}$ to the predicate $P : \mathsf{list}\ A_0 \to \cdots \to \mathsf{list}\ A_{n-1} \to \mathsf{Prop}$ that guarantees that P holds for all (pairs of) elements of the list(s). The lifted predicate requires all lists to have the same length in case $n > 1$.
- We let $|\vec{x}| \in \mathbb{N}$ denote the length of $\vec{x} \in \mathsf{list}\ A$.
- We let $\vec{x}_{[i,\,j)} \in \mathsf{list}\ A$ denote the sublist $x_i \ldots x_{j-1}$ of $\vec{x} \in \mathsf{list}\ A$.
- We let $x^n \in \mathsf{list}\ A$ denote the list consisting of n times $x \in A$.
- We let $(\vec{x}y^\infty)_{[i,\,j)} \in \mathsf{list}\ A$ denote the sublist $x_i \ldots x_{j-1}$ of $\vec{x} \in \mathsf{list}\ A$ which is padded with $y \in A$ in case $\vec{x}$ is too short.
- Given lists $\vec{x} \in \mathsf{list}\ A$ and $\vec{y} \in \mathsf{list}\ B$ with $|\vec{x}| = |\vec{y}|$, we let $\overrightarrow{x\vec{y}} \in \mathsf{list}\ (A \times B)$ denote the point-wise pairing of $\vec{x}$ and $\vec{y}$.

3 Challenges

This section illustrates a number of subtle forms of underspecification in C by means of example programs, their bizarre behaviors exhibited by widely used C compilers, and their treatment in $\mathrm{CH_2O}$. Many of these examples involve delicacies due to the interaction between the following two ways of accessing data:

- In a *high-level* way using arrays, structs and unions.

– In a *low-level* way using unstructured and untyped byte representations.

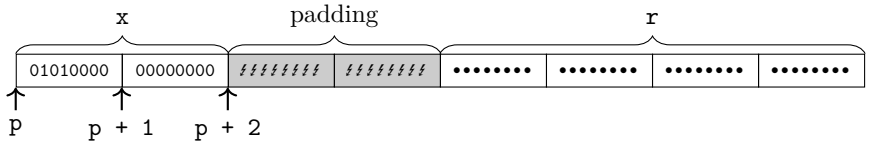
The main problem is that compilers use a high-level view of data access to perform optimizations whereas both programmers and traditional memory models expect data access to behave in a concrete low-level way.

3.1 Byte-level operations and object representations

Apart from *high-level* access to objects in memory by means of typed expressions, C also allows *low-level* access by means of byte-wise manipulation. Each object of type τ can be interpreted as an `unsigned char` array of length `sizeof`(τ), which is called the *object representation* [27, 6.2.6.1p4]. Let us consider:

```
struct S { short x; short *r; } s1 = { 10, &s1.x };
unsigned char *p = (unsigned char*)&s1;
```

On 32-bit computing architectures such as x86 (with `_Alignof(short*)`$= 4$), the object representation of `s1` might be:



The above object representation contains a hole due to *alignment* of objects. The bytes belonging to such holes are called *padding bytes*.

Alignment is the way objects are arranged in memory. In modern computing architectures, accesses to addresses that are a multiple of a word sized chunk (often a multiple of 4 bytes on a 32-bit computing architecture) are significantly faster due to the way the processor interacts with the memory. For that reason, the C11 standard has put restrictions on the addresses at which objects may be allocated [27, 6.2.8]. For each type τ, there is an implementation-defined integer constant `_Alignof(`τ`)`, and objects of type τ are required to be allocated at addresses that are a multiple of that constant. In case `_Alignof(short*)`$= 4$, there are thus two bytes of padding in between the fields of `struct S`.

An object can be copied by copying its object representation. For example, the struct `s1` can be copied to `s2` as follows:

```
struct S { short x; short *r; } s1 = { 10, &s1.x };
struct S s2;
for (size_t i = 0; i < sizeof(struct S); i++)
  ((unsigned char*)&s2)[i] = ((unsigned char*)&s1)[i];
```

In the above code, `size_t` is an unsigned integer type, which is able to hold the results of the `sizeof` operator [27, 7.19p2].

Manipulation of object representations of structs also involves access to padding bytes, which are not part of the high-level representation. In particular, in the example the padding bytes are also being copied. The problematic part is that padding bytes have indeterminate values, whereas in general, reading an indeterminate value

has undefined behavior (for example, reading from an uninitialized `int` variable is undefined). The C11 standard provides an exception for `unsigned char` [27, 6.2.6.1p5], and the above example thus has defined behavior.

Our memory model uses a symbolic representation of bits (Definition 6.19) to distinguish determinate and indeterminate memory. This way, we can precisely keep track of the situations in which access to indeterminate memory is permitted.

3.2 Padding of structs and unions

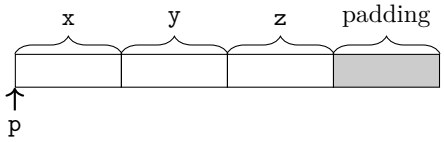
The following excerpt from the C11 standard points out another challenge with respect to padding bytes [27, 6.2.6.1p6]:

> When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values.

Let us illustrate this difficulty by an example:

```
struct S { char x; char y; char z; };
void f(struct S *p) { p->x = 0; p->y = 0; p->z = 0; }
```

On architectures with `sizeof(struct S) = 4`, objects of type `struct S` have one byte of padding. The object representation may be as follows:



Instead of compiling the function `f` to three store instructions for each field of the struct, the C11 standard allows a compiler to use a single instruction to store zeros to the entire struct. This will of course affect the padding byte. Consider:

```
struct S s = { 1, 1, 1 };
((unsigned char*)&s)[3] = 10;
f(&s);
return ((unsigned char*)&s)[3];
```

Now, the assignments to fields of `s` by the function `f` affect also the padding bytes of `s`, including the one `((unsigned char*)&s)[3]` that we have assigned to. As a consequence, the returned value is unspecified.

From a high-level perspective this behavior makes sense. Padding bytes are not part of the abstract value of a struct, so their actual value should not matter. However, from a low-level perspective it is peculiar. An assignment to a specific field of a struct affects the object representation of parts not assigned to.

None of the currently existing C formalizations describes this behavior correctly. In our tree based memory model we enforce that padding bytes always have an indeterminate value, and in turn we have the desired behavior implicitly. Note that if the function call `f(&s)` would have been removed, the behavior of the example program remains unchanged in CH_2O.

3.3 Type-punning

Despite the rules for effective types, it is under certain conditions nonetheless allowed to access a union through another variant than the current one. Accessing a union through another variant is called *type-punning*. For example:

```
union int_or_short { int x; short y; } u = { .x = 3 };
printf("%d\n", u.y);
```

This code will reinterpret the bit representation of the `int` value `3` of `u.x` as a value of type `short`. The reinterpreted value that is printed is implementation-defined (on architectures where `short`s do not have trap values).

Since C11 is ambiguous about the exact conditions under which type-punning is allowed[1], we follow the interpretation by the GCC documentation [20]:

> Type-punning is allowed, provided the memory is accessed through the union type.

According to this interpretation the above program indeed has implementation defined behavior because the variant `y` is accessed via the expression `u.y` that involves the variable `u` of the corresponding union type.

However, according to this interpretation, type-punning via a pointer to a specific variant of a union type yields undefined behavior. This is in agreement with the rules for effective types. For example, the following program has undefined behavior.

```
union int_or_short { int x; short y; } u = { .x = 3 };
short *p = &u.y;
printf("%d\n", *p);
```

We formalize the interpretation of C11 by GCC by decorating pointers and l-values to subobjects with annotations (Definition 6.4). When a pointer to a variant of a union is stored in memory, or used as the argument of a function, the annotations are changed to ensure that type-punning no longer has defined behavior via that pointer. In Section 7.1 we formally establish that this approach is correct by showing that a compiler can perform type-based alias analysis (Theorem 7.2 on page 51).

3.4 Indeterminate memory and pointers

A pointer value becomes indeterminate when the object it points to has reached the end of its lifetime [27, 6.2.4] (it has gone out of scope, or has been deallocated). Dereferencing an indeterminate pointer has of course undefined behavior because it no longer points to an actual value. However, not many people are aware that using an indeterminate pointer in pointer arithmetic and pointer comparisons also yields undefined behavior. Consider:

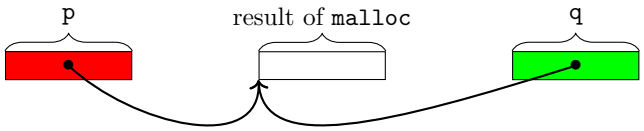
```
int *p = malloc(sizeof(int)); assert (p != NULL);
free(p);
```

[1] The term *type-punning* merely appears in a footnote [27, footnote 95]. There is however the related *common initial sequence* rule [27, 6.5.2.3], for which the C11 standard uses the notion of *visible*. This notion is not clearly defined either.

```
int *q = malloc(sizeof(int)); assert (q != NULL);
if (p == q) { // undefined, p is indeterminate due to the free
  *q = 10;
  *p = 14;
  printf("%d\n", *q); // p and q alias, expected to print 14
}
```

In this code `malloc(sizeof(int))` yields a pointer to a newly allocated memory area that may hold an integer, or yields a `NULL` pointer in case no memory is available. The function `free` deallocates memory allocated by `malloc`. In the example we assert that both calls to `malloc` succeed.

After execution of the second call to `malloc` it may happen that the memory area of the first call to `malloc` is reused: we have used `free` to deallocate it after all. This would lead to the following situation in memory:



Both GCC (version 4.9.2) or Clang (version 3.5.0) use the fact that `p` and `q` are obtained via different calls to `malloc` as a license to assume that `p` and `q` do not alias. As a result, the value 10 of `*q` is inlined, and the program prints the value 10 instead of the naively expected value 14.

The situation becomes more subtle because when the object a pointer points to has been deallocated, not just the argument of `free` becomes indeterminate, but also all other copies of that pointer. This is therefore yet another example where high-level representations interact subtly with their low-level counterparts.

In our memory model we represent pointer values symbolically (Definition 6.4), and keep track of memory areas that have been previously deallocated. The behavior of operations like `==` depends on the memory state, which allows us to accurately capture the described undefined behaviors.

3.5 End-of-array pointers

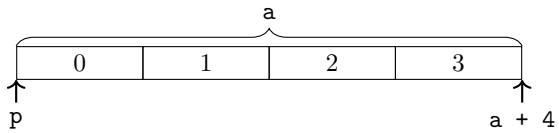
The way the C11 standard deals with pointer equality is subtle. Consider the following excerpt [27, 6.5.9p6]:

> Two pointers compare equal if and only if [...] or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space.

End-of-array pointers are peculiar because they cannot be dereferenced, they do not point to any value after all. Nonetheless, end-of-array are commonly used when looping through arrays.

```
int a[4] = { 0, 1, 2, 3 };
int *p = a;
while (p < a + 4) { *p += 1; p += 1; }
```

The pointer `p` initially refers to the first element of the array `a`. The value `p` points to, as well as `p` itself, is being increased as long as `p` is before the end-of-array pointer `a + 4`. This code thus increases the values of the array `a`. The initial state of the memory is displayed below:



End-of-array pointers can also be used in a way where the result of a comparison is not well-defined. In the example below, the `printf` is executed only if `x` and `y` are allocated adjacently in the address space (typically the stack).

```
int x, y;
if (&x + 1 == &y) printf("x and y are allocated adjacently\n");
```

Based on the aforementioned excerpt of the C11 standard [27, 6.5.9p6], one would naively say that the value of `&x + 1 == &y` is uniquely determined by the way `x` and `y` are allocated in the address space. However, the GCC implementers disagree[2]. They claim that Defect Report #260 [26] allows them to take the *derivation of a pointer value* into account.

In the example, the pointers `&x + 1` and `&y` are derived from unrelated objects (the local variables `x` and `y`). As a result, the GCC developers claim that `&x + 1` and `&y` may compare unequal albeit being allocated adjacently. Consider:

```
int compare(int *p, int *q) {
  // some code to confuse the optimizer
  return p == q;
}
int main() {
  int x, y;
  if (&x + 1 == &y) printf("x and y are adjacent\n");
  if (compare(&x + 1, &y)) printf("x and y are still adjacent\n");
}
```

When compiled with GCC (version 4.9.2), we have observed that the string `x and y are still adjacent` is being printed, wheras `x and y are adjacent` is not being printed. This means that the value of `&x + 1 == &y` is not consistent among different occurrences of the comparison.

Due to these discrepancies we assign undefined behavior to questionable uses of end-of-array pointers while assigning the correct defined behavior to pointer comparisons involving end-of-array pointers when looping through arrays (such as in the first example above). Our treatment is similar to our extension of CompCert [34].

3.6 Sequence point violations and non-determinism

Instead of having to follow a specific execution order, the execution order of expressions is unspecified in C. This is a common cause of portability problems because

[2] See `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=61502`

a compiler may use an arbitrary execution order for each expression, and each time that expression is executed. Hence, to ensure correctness of a C program with respect to an arbitrary compiler, one has to verify that *each* possible execution order is free of undefined behavior and gives the correct result.

In order to make more effective optimizations possible (for example, delaying of side-effects and interleaving), the C standard does not allow an object to be modified more than once during the execution of an expression. If an object is modified more than once, the program has undefined behavior. We call this requirement the *sequence point restriction*. Note that this is not a static restriction, but a restriction on valid executions of the program. Let us consider an example:

```
int x, y = (x = 3) + (x = 4);
printf("%d %d\n", x, y);
```

By considering all possible execution orders, one would naively expect this program to print `4 7` or `3 7`, depending on whether the assignment `x = 3` or `x = 4` is executed first. However, `x` is modified twice within the same expression, and thus both execution orders have undefined behavior. The program is thereby allowed to exhibit any behavior. Indeed, when compiled with `gcc -O2` (version 4.9.2), the compiled program prints `4 8`, which does not correspond to any of the execution orders.

Our approach to non-determinism and sequence points is inspired by Norrish [44] and Ellison and Roşu [19]. Each bit in memory carries a permission (Definition 5.5) that is set to a special *locked* permission when a store has been performed. The memory model prohibits any access (read or store) to objects with locked permissions. At the next sequence point, the permissions of locked objects are changed back into their original permission, making future accesses possible again.

It is important to note that we do not have non-determinism in the memory model itself, and have set up the memory model in such a way that all non-determinism is on the level of the small-step operational semantics.

4 Types in C

This section describes the types used in the CH_2O memory model. We support integer, pointer, function pointer, array, struct, union and void types. More complicated types such as enum types and typedefs are defined by translation [33, 37].

This section furthermore describes an abstract interface, called an *implementation environment*, that describes properties such as size and endianness of integers, and the layout of structs and unions. The entire CH_2O memory model and semantics will be parameterized by an implementation environment.

4.1 Integer representations

This section describes the part of implementation environments corresponding to integer types and the encoding of integer values as bits. Integer types consist of a *rank* (char, short, int ...) and a *signedness* (signed or unsigned). The set of available ranks as well as many of their properties are implementation-defined. We therefore abstract over the ranks in the definition of integer types.

Definition 4.1 *Integer signedness* and *integer types* over ranks $k \in K$ are inductively defined as:

$$si \in \mathsf{signedness} ::= \mathsf{signed} \mid \mathsf{unsigned}$$
$$\tau_\mathsf{i} \in \mathsf{inttype} ::= si\ k$$

The projections are called $\mathsf{rank} : \mathsf{inttype} \to K$ and $\mathsf{sign} : \mathsf{inttype} \to \mathsf{signedness}$.

Definition 4.2 An *integer coding environment with ranks* K consists of a total order $(K, \subseteq)$ of *integer ranks* having at least the following ranks:

$$\mathsf{char} \subset \mathsf{short} \subset \mathsf{int} \subset \mathsf{long} \subset \mathsf{long\ long} \qquad \text{and} \qquad \mathsf{ptr\_rank}.$$

It moreover has the following functions:

$$\mathsf{char\_bits} : \mathbb{N}_{\geq 8} \qquad \mathsf{endianize} : K \to \mathsf{list\ bool} \to \mathsf{list\ bool}$$
$$\mathsf{char\_signedness} : \mathsf{signedness} \qquad \mathsf{deendianize} : K \to \mathsf{list\ bool} \to \mathsf{list\ bool}$$
$$\mathsf{rank\_size} : K \to \mathbb{N}_{>0}$$

Here, $\mathsf{endianize}\ k$ and $\mathsf{deendianize}\ k$ should be inverses, $\mathsf{endianize}\ k$ should be a permutation, $\mathsf{rank\_size}$ should be (non-strictly) monotone, and $\mathsf{rank\_size}\ \mathsf{char} = 1$.

Definition 4.3 The judgment $x : \tau_\mathsf{i}$ describes that $x \in \mathbb{Z}$ *has integer type* τ_i.

$$\frac{-2^{\mathsf{char\_bits} * \mathsf{rank\_size}\ k-1} \leq x < 2^{\mathsf{char\_bits} * \mathsf{rank\_size}\ k-1}}{x : \mathsf{signed}\ k} \qquad \frac{0 \leq x < 2^{\mathsf{char\_bits} * \mathsf{rank\_size}\ k}}{x : \mathsf{unsigned}\ k}$$

The rank char is the rank of the smallest integer type, whose unsigned variant corresponds to bytes that constitute object representations (see Section 3.1). Its bit size is $\mathsf{char\_bits}$ (called `CHAR_BIT` in the standard library header files [27, 5.2.4.2.1]), and its signedness $\mathsf{char\_signedness}$ is implementation-defined [27, 6.2.5p15].

The rank $\mathsf{ptr\_rank}$ is the rank of the integer types `size_t` and `ptrdiff_t`, which are defined in the standard library header files [27, 7.19p2]. The type `ptrdiff_t` is a signed integer type used to represent the result of subtracting two pointers, and the type `size_t` is an unsigned integer type used to represent sizes of types.

An integer coding environment can have an arbitrary number of integer ranks apart from the standard ones char, short, int, long, $\mathsf{long\ long}$, and $\mathsf{ptr\_rank}$. This way, additional integer types like those describe in [27, 7.20] can easily be included.

The function $\mathsf{rank\_size}$ gives the byte size of an integer of a given rank. Since we require $\mathsf{rank\_size}$ to be monotone rather than strictly monotone, integer types with different ranks can have the same size [27, 6.3.1.1p1]. For example, on many implementations `int` and `long` have the same size, but are in fact different.

The C11 standard allows implementations to use either sign-magnitude, 1's complement or 2's complement signed integers representations. It moreover allows integer representations to contain padding or parity bits [27, 6.2.6.2]. However, since all current machine architectures use 2's complement representations, this is more of a historic artifact. Current machine architectures use 2's complement representations because these do not suffer from positive and negative zeros and thus enjoy unique representations of the same integer. Hence, $\mathrm{CH_2O}$ restricts itself to implementations that use 2's complement signed integers representations.

Integer representations in CH_2O can solely differ with respect to endianness (the order of the bits). The function endianize takes a list of bits in little endian order and permutes them accordingly. We allow endianize to yield an arbitrary permutation and thus we not just support big- and little-endian, but also mixed-endian variants.

Definition 4.4 Given an integer type τ_i, the *integer encoding functions* $\overline{\_ : \tau_\mathsf{i}} : \mathbb{Z} \to \mathsf{list\ bool}$ and $(\_)_{\tau_\mathsf{i}} : \mathsf{list\ bool} \to \mathbb{Z}$ are defined as follows:

$$\overline{x : si\ k} := \mathsf{endianize}\ k\ (x \text{ as little endian 2's complement})$$
$$(\vec{\beta})_{si\ k} := \text{of little endian 2's complement } (\mathsf{deendianize}\ k\ \vec{\beta})$$

Lemma 4.5 *The integer encoding functions are inverses. That means:*

1. *We have* $(\overline{x : \tau_\mathsf{i}})_{\tau_\mathsf{i}} = x$ *and* $|\overline{x : \tau_\mathsf{i}}| = \mathsf{rank\_size}\ \tau_\mathsf{i}$ *provided that* $x : \tau_\mathsf{i}$.
2. *We have* $\overline{(\vec{\beta})_{\tau_\mathsf{i}} : \tau_\mathsf{i}} = \vec{\beta}$ *and* $(\vec{\beta})_{\tau_\mathsf{i}} : \tau_\mathsf{i}$ *provided that* $|\vec{\beta}| = \mathsf{rank\_size}\ \tau_\mathsf{i}$.

4.2 Definition of types

We support integer, pointer, function pointer, array, struct, union and void types. The translation that we have described in [33, 37] translates more complicated types, such as typedefs and enums, into these simplified types. This translation also alleviates other simplifications of our simplified definition of types, such as the use of unnamed struct and union fields. Floating point types and type qualifiers like `const` and `volatile` are not supported.

All definitions in this section are implicitly parameterized by an integer coding environment with ranks K (Definition 4.2).

Definition 4.6 *Tags* $t \in \mathsf{tag}$ (sometimes called *struct/union names*) and *function names* $f \in \mathsf{funname}$ are represented as strings.

Definition 4.7 *Types* consist of *point-to types*, *base types* and *full types*. These are inductively defined as:

$$\tau_\mathsf{p}, \sigma_\mathsf{p} \in \mathsf{ptrtype} ::= \tau \mid \mathsf{any} \mid \vec{\tau} \to \tau$$
$$\tau_\mathsf{b}, \sigma_\mathsf{b} \in \mathsf{basetype} ::= \tau_\mathsf{i} \mid \tau_\mathsf{p}* \mid \mathsf{void}$$
$$\tau, \sigma \in \mathsf{type} ::= \tau_\mathsf{b} \mid \tau[n] \mid \mathsf{struct}\ t \mid \mathsf{union}\ t$$

The three mutual inductive parts of types correspond to the different components of the memory model. Addresses and pointers have point-to types (Definitions 6.8 and 6.10), base values have base types (Definition 6.40), and memory trees and values have full types (Definitions 6.25 and 6.46).

The void type of C is used for two entirely unrelated purposes: `void` is used for functions without return type and `void*` is used for pointers to objects of unspecified type. In CH_2O this distinction is explicit in the syntax of types. The type void is used for function without return value. Like the mathematical *unit* type it has one value called nothing (Definition 6.39). The type any$*$ is used for pointers to objects of unspecified type.

Unlike more modern programming languages C does not provide first class functions. Instead, C provides function pointers which are just addresses of executable

code in memory instead of closures. Function pointers can be used in a way similar to ordinary pointers: they can be used as arguments and return value of functions, they can be part of structs, unions and arrays, *etc.*

The C language sometimes allows function types to be used as shorthands for function pointers, for example:

```
void sort(int *p, int len, int compare(int,int));
```

The third argument of `sort` is a shorthand for `int (*compare)(int,int)` and is thus in fact a function pointer instead of a function. We only have function pointer types, and the third argument of the type of the function `sort` thus contains an additional $*$:

$$[\,(\mathsf{signed\ int})*,\ \mathsf{signed\ int},\ (\mathsf{signed\ int} \to \mathsf{signed\ int})*\,] \to \mathsf{void}.$$

Struct and union types consist of just a name, and do not contain the types of their fields. An environment is used to assign fields to structs and unions, and to assign argument and return types to function names.

Definition 4.8 *Type environments* are defined as:

$$\begin{aligned}\Gamma \in \mathsf{env} := {} & (\mathsf{tag} \to_{\mathsf{fin}} \mathsf{list\ type}) \times && \text{(types of struct/union fields)}\\ & (\mathsf{funname} \to_{\mathsf{fin}} (\mathsf{list\ type} \times \mathsf{type})) && \text{(types of functions)}\end{aligned}$$

The functions $\mathsf{dom}_{\mathsf{tag}} : \mathsf{env} \to \mathcal{P}_{\mathsf{fin}}(\mathsf{tag})$ and $\mathsf{dom}_{\mathsf{funname}} : \mathsf{env} \to \mathcal{P}_{\mathsf{fin}}(\mathsf{funname})$ yield the declared structs and unions, respectively the declared functions. We implicitly treat environments as functions $\mathsf{tag} \to_{\mathsf{fin}} \mathsf{list\ type}$ and $\mathsf{funname} \to_{\mathsf{fin}} (\mathsf{list\ type} \times \mathsf{type})$ that correspond to underlying finite partial functions.

Struct and union names on the one hand, and function names on the other, have their own name space in accordance with the C11 standard [27, 6.2.3p1].

Notation 4.9 We often write an environment as a mixed sequence of struct and union declarations $t : \vec{\tau}$, and function declarations $f : (\vec{\tau}, \tau)$. This is possible because environments are finite.

Since we represent the fields of structs and unions as lists, fields are nameless. For example, the C type `struct S1 { int x; struct S1 *p; }` is translated into the environment $\mathtt{S1} : [\,\mathsf{signed\ int}, \mathsf{struct\ S1}*\,]$.

Although structs and unions are semantically very different (products versus sums, respectively), environments do not keep track of whether a tag has been used for a struct or a union type. Structs and union types with the same tag are thus allowed. The translator in [33, 37] forbids the same name being used to declare both a struct and union type.

Although our mutual inductive syntax of types already forbids many incorrect types such as functions returning functions (instead of function pointers), still some ill-formed types such as `int[0]` are syntactically valid. Also, we have to ensure that cyclic structs and unions are only allowed when the recursive definition is guarded through pointers. Guardedness by pointers ensures that the sizes of types are finite and statically known. Consider the following types:

```
struct list1 { int hd; struct list1 tl; };     /* illegal */
struct list2 { int hd; struct list2 *p_tl; };  /* legal */
```

The type declaration **struct list1** is illegal because it has a reference to itself. In the type declaration **struct list2** the self reference is guarded through a pointer type, and therefore legal. Of course, this generalizes to mutual recursive types like:

```
struct tree { int hd; struct forest *p_children; };
struct forest { struct tree *p_hd; struct forest *p_tl; };
```

Definition 4.10 The following judgments are defined by mutual induction:

– The judgment $\Gamma \vdash_* \tau_\mathsf{p}$ describes *point-to types* τ_p *to which a pointer may point*:

$$\frac{}{\Gamma \vdash_* \mathsf{any}} \qquad \frac{\Gamma \vdash_* \vec{\tau} \qquad \Gamma \vdash_* \tau}{\Gamma \vdash_* \vec{\tau} \to \tau} \qquad \frac{\Gamma \vdash_\mathsf{b} \tau_\mathsf{b}}{\Gamma \vdash_* \tau_\mathsf{b}} \qquad \frac{\Gamma \vdash \tau \qquad n \neq 0}{\Gamma \vdash_* \tau[n]}$$

$$\frac{}{\Gamma \vdash_* \mathsf{struct}\ t} \qquad \frac{}{\Gamma \vdash_* \mathsf{union}\ t}$$

– The judgment $\Gamma \vdash_\mathsf{b} \tau_\mathsf{b}$ describes *valid base types* τ_b:

$$\frac{}{\Gamma \vdash_\mathsf{b} \tau_\mathsf{i}} \qquad \frac{\Gamma \vdash_* \tau_\mathsf{p}}{\Gamma \vdash_\mathsf{b} \tau_\mathsf{p}*} \qquad \frac{}{\Gamma \vdash_\mathsf{b} \mathsf{void}}$$

– The judgment $\Gamma \vdash \tau$ describes *valid types* τ:

$$\frac{\Gamma \vdash_\mathsf{b} \tau_\mathsf{b}}{\Gamma \vdash \tau_\mathsf{b}} \qquad \frac{\Gamma \vdash \tau \qquad n \neq 0}{\Gamma \vdash \tau[n]} \qquad \frac{t \in \mathsf{dom}_\mathsf{tag}\ \Gamma}{\Gamma \vdash \mathsf{struct}\ t} \qquad \frac{t \in \mathsf{dom}_\mathsf{tag}\ \Gamma}{\Gamma \vdash \mathsf{union}\ t}$$

Definition 4.11 The judgment $\vdash \Gamma$ describes *well-formed environments* Γ. It is inductively defined as:

$$\frac{}{\vdash \emptyset} \qquad \frac{\vdash \Gamma \quad \Gamma \vdash \vec{\tau} \quad \vec{\tau} \neq \varepsilon \quad t \notin \mathsf{dom}_\mathsf{tag}\ \Gamma}{\vdash t : \vec{\tau}, \Gamma} \qquad \frac{\vdash \Gamma \quad \Gamma \vdash_* \vec{\tau} \quad \Gamma \vdash_* \tau \quad f \notin \mathsf{dom}_\mathsf{funname}\ \Gamma}{\vdash f : (\vec{\tau}, \tau), \Gamma}$$

Note that $\Gamma \vdash \tau$ does not imply $\vdash \Gamma$. Most results therefore have $\vdash \Gamma$ as a premise. These premises are left implicit in this paper.

In order to support (mutually) recursive struct and union types, pointers to incomplete struct and union types are permitted in the judgment $\Gamma \vdash_* \tau_\mathsf{p}$ that describes types to which pointers are allowed, but forbidden in the judgment $\Gamma \vdash \tau$ of validity of types. Let us consider the following type declarations:

```
struct S2 { struct S2 x; };   /* illegal */
struct S3 { struct S3 *p; };  /* legal */
```

Well-formedness $\vdash \Gamma$ of the environment $\Gamma := \mathtt{S3} : [\,\mathsf{struct}\ \mathtt{S3}*\,]$ can be derived using the judgments $\emptyset \vdash_* \mathsf{struct}\ \mathtt{S3}$, $\emptyset \vdash_\mathsf{b} \mathsf{struct}\ \mathtt{S3}*$, $\emptyset \vdash \mathsf{struct}\ \mathtt{S3}*$, and thus $\vdash \Gamma$. The environment $\mathtt{S2} : [\,\mathsf{struct}\ \mathtt{S2}\,]$ is ill-formed because we do not have $\emptyset \vdash \mathsf{struct}\ \mathtt{S2}$.

The typing rule for function pointers types is slightly more delicate. This is best illustrated by an example:

```
union U { int i; union U (*f) (union U); };
```

This example displays a recursive self reference to a union type through a function type, which is legal in C because function types are in fact pointer types. Due to this reason, the premises of $\Gamma \vdash_* \vec{\tau} \to \tau$ are $\Gamma \vdash_* \vec{\tau}$ and $\Gamma \vdash_* \tau$ instead of $\Gamma \vdash \vec{\tau}$ and $\Gamma \vdash \tau$. Well-formedness of the above union type can be derived as follows:

$$\dfrac{\vdash \Gamma \qquad \dfrac{\dfrac{}{\Gamma \vdash_\mathsf{b} \mathsf{signed\ int}}}{\Gamma \vdash \mathsf{signed\ int}} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash_* \mathsf{union\ U}} \qquad \dfrac{}{\Gamma \vdash_* \mathsf{union\ U}}}{\Gamma \vdash_* \mathsf{union\ U} \to \mathsf{union\ U}}}{\Gamma \vdash_\mathsf{b} (\mathsf{union\ U} \to \mathsf{union\ U})*}}{\Gamma \vdash (\mathsf{union\ U} \to \mathsf{union\ U})*}}{\vdash \mathtt{U} : [\,\mathsf{signed\ int}, (\mathsf{union\ U} \to \mathsf{union\ U})*\,], \Gamma}$$

In order to define operations by recursion over the structure of well-formed types (see for example Definition 6.45, which turns a sequence of bits into a value), we often need to perform recursive calls on the types of fields of structs and unions. In Coq we have defined a custom recursor and induction principle using well-founded recursion. In this paper, we will use these implicitly.

Affeldt *et al.* [1, 2] have formalized non-cyclicity of types using a complex constraint on paths through types. Our definition of validity of environments (Definition 4.11) follows the structure of type environments, and is therefore well-suited to implement the aforementioned recursor and induction principle.

There is a close correspondence between array and pointer types in C. Arrays are not first class types, and except for special cases such as initialization, manipulation of arrays is achieved via pointers. We consider arrays as first class types so as to avoid having to make exceptions for the case of arrays all the time.

Due to this reason, more types are valid in $\mathrm{CH_2O}$ than in C11. The translator in [33, 37] resolves exceptional cases for arrays. For example, a function parameter of array type acts like a parameter of pointer type in C11 [27, 6.7.6.3][3].

```
void f(int a[10]);
```

The corresponding type of the function `f` is thus $(\mathsf{signed\ int})* \to \mathsf{void}$. Note that the type $(\mathsf{signed\ int})[10] \to \mathsf{void}$ is also valid, but entirely different, and never generated by the translator in [33, 37].

4.3 Implementation environments

We finish this section by extending integer coding environments to describe implementation-defined properties related the layout of struct and union types. The author's PhD thesis [33] also considers the implementation-defined behavior of integer operations (such as addition and division) and defines inhabitants of this interface corresponding to actual computing architectures.

Definition 4.12 A *implementation environment with ranks* K consists of an integer coding environment with ranks K and functions:

$$\mathsf{sizeof}_\Gamma : \mathsf{type} \to \mathbb{N} \qquad \mathsf{alignof}_\Gamma : \mathsf{type} \to \mathbb{N} \qquad \mathsf{fieldsizes}_\Gamma : \mathsf{list\ type} \to \mathsf{list}\ \mathbb{N}$$

[3] The array size is ignored unless the `static` keyword is used. In case `f` would have the prototype `void f(int a[static 10])`, the pointer `a` should provide access to an array of at least 10 elements [27, 6.7.6.3]. The `static` keyword is not supported by $\mathrm{CH_2O}$.

These functions should satisfy:

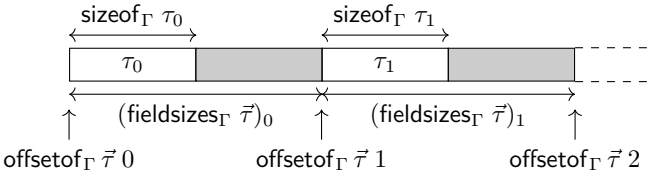
$$\begin{aligned}
\mathsf{sizeof}_\Gamma\ (si\ k) &= \mathsf{rank_size}\ k && \mathsf{sizeof}_\Gamma\ (\tau_\mathsf{p}*) \neq 0 \qquad \mathsf{sizeof}_\Gamma\ \mathsf{void} \neq 0\\
\mathsf{sizeof}_\Gamma\ (\tau[n]) &= n * \mathsf{sizeof}_\Gamma\ \tau\\
\mathsf{sizeof}_\Gamma\ (\mathsf{struct}\ t) &= \Sigma\ \mathsf{fieldsizes}_\Gamma\ \vec{\tau} && \text{if } \Gamma\, t = \vec{\tau}\\
\mathsf{sizeof}_\Gamma\ \tau_i &\leq z_i \text{ and } |\vec{\tau}| = |\vec{z}| && \text{if } \mathsf{fieldsizes}_\Gamma\ \vec{\tau} = \vec{z}, \text{ for each } i < |\vec{\tau}|\\
\mathsf{sizeof}_\Gamma\ \tau_i &\leq \mathsf{sizeof}_\Gamma\ (\mathsf{union}\ t) && \text{if } \Gamma\, t = \vec{\tau}, \text{ for each } i < |\vec{\tau}|\\
\mathsf{alignof}_\Gamma\ \tau &\mid \mathsf{alignof}_\Gamma\ (\tau[n])\\
\mathsf{alignof}_\Gamma\ \tau_i &\mid \mathsf{alignof}_\Gamma\ (\mathsf{struct}\ t) && \text{if } \Gamma\, t = \vec{\tau}, \text{ for each } i < |\vec{\tau}|\\
\mathsf{alignof}_\Gamma\ \tau_i &\mid \mathsf{alignof}_\Gamma\ (\mathsf{union}\ t) && \text{if } \Gamma\, t = \vec{\tau}, \text{ for each } i < |\vec{\tau}|\\
\mathsf{alignof}_\Gamma\ \tau &\mid \mathsf{sizeof}_\Gamma\ \tau && \text{if } \Gamma \vdash \tau\\
\mathsf{alignof}_\Gamma\ \tau_i &\mid \mathsf{offsetof}_\Gamma\ \vec{\tau}\ i && \text{if } \Gamma \vdash \vec{\tau}, \text{ for each } i < |\vec{\tau}|
\end{aligned}$$

Here, we let $\mathsf{offsetof}_\Gamma\ \vec{\tau}\ i$ denote $\Sigma_{j<i}(\mathsf{fieldsizes}_\Gamma\ \vec{\tau})_j$. The functions sizeof_Γ, $\mathsf{alignof}_\Gamma$, and $\mathsf{fieldsizes}_\Gamma$ should be closed under weakening of Γ.

Notation 4.13 Given an implementation environment, we let:

$$\begin{aligned}
\mathsf{bitsizeof}_\Gamma\ \tau &:= \mathsf{sizeof}_\Gamma\ \tau \cdot \mathsf{char_bits}\\
\mathsf{bitoffsetof}_\Gamma\ \tau\ j &:= \mathsf{offsetof}_\Gamma\ \tau\ j \cdot \mathsf{char_bits}\\
\mathsf{fieldbitsizes}_\Gamma\ \tau &:= \mathsf{fieldsizes}_\Gamma\ \tau \cdot \mathsf{char_bits}
\end{aligned}$$

We let $\mathsf{sizeof}_\Gamma\ \tau$ specify the number of bytes out of which the object representation of an object of type τ consists. Objects of type τ should be allocated at addresses that are a multiple of $\mathsf{alignof}_\Gamma\ \tau$. We will prove that our abstract notion of addresses satisfies this property (see Lemma 6.18). The functions sizeof_Γ, $\mathsf{alignof}_\Gamma$ correspond to the `sizeof` and `_Alignof` operators [27, 6.5.3.4], and $\mathsf{offsetof}_\Gamma$ corresponds to the `offsetof` macro [27, 7.19p3]. The list $\mathsf{fieldsizes}_\Gamma\ \vec{\tau}$ specifies the layout of a struct type with fields $\vec{\tau}$ as follows:



5 Permissions and separation algebras

Permissions control whether memory operations such as a read or store are allowed or not. In order to obtain the highest level of precision, we tag each individual bit in memory with a corresponding permission. In the operational semantics, permissions have two main purposes:

– Permissions are used to formalize the *sequence point restriction* which assigns undefined behavior to programs in which an object in memory is modified more than once in between two sequence points.

– Permissions are used to distinguish objects in memory that are writable from those that are read-only (*const qualified* in C terminology).

In the axiomatic semantics based on separation logic, permissions play an important role for *share accounting*. We use share accounting for *subdivision of permissions* among multiple subexpressions to ensure that:

– Writable objects are unique to each subexpression.
– Read-only objects may be shared between subexpressions.

This distinction is originally due to Dijkstra [16] and is essential in separation logic with permissions [11]. The novelty of our work is to use separation logic with permissions for non-determinism in expressions in C. Share accounting gives rise to a natural treatment of C's sequence point restriction.

Separation algebras as introduced by Calcagno *et al.* [13] abstractly capture common structure of subdivision of permissions. We present a generalization of separation algebras that is well-suited for C verification in Coq and use this generalization to build the permission system and memory model compositionally. The permission system will be constructed as a telescope of separation algebras:

$$\mathsf{perm} \quad := \quad \underbrace{\mathcal{L}(\mathcal{C}(\mathbb{Q}))}_{\text{non-const qualified}} \quad + \quad \underbrace{\mathbb{Q}}_{\text{const qualified}}$$

Here, $\mathbb{Q}$ is the separation algebra of fractional permissions, $\mathcal{C}$ is a functor that extends a separation algebra with a counting component, and $\mathcal{L}$ is a functor that extends a separation algebra with a lockable component (used for the sequence point restriction). This section explains these functors and their purposes.

5.1 Separation logic and share accounting

Before we will go into the details of the $\mathrm{CH_2O}$ permission system, we briefly introduce separation logic. Separation logic [47] is an extension of Hoare logic that provides better means to reason about imperative programs that use mutable data structures and pointers. The key feature of separation logic is the *separating conjunction* $P * Q$ that allows one to subdivide the memory into two disjoint parts: a part described by P and another part described by Q. The separating conjunction is most prominent in the *frame rule*.

$$\frac{\{P\}\, s\, \{Q\}}{\{P * R\}\, s\, \{Q * R\}}$$

This rule enables local reasoning. Given a Hoare triple $\{P\}\, s\, \{Q\}$, this rule allows one to derive that the triple also holds when the memory is extended with a disjoint part described by R. The frame rule shows its merits when reasoning about functions. There it allows one to consider a function in the context of the memory the function actually uses, instead of having to consider the function in the context of the entire program's memory. However, already in derivations of small programs the use of the frame rule can be demonstrated[4]:

[4] Contrary to traditional separation logic, we do not give local variables a special status of being *stack allocated*. We do so, because in C even local variables are allowed to have pointers to them.

$$\dfrac{\dfrac{\overline{\{\mathtt{x} \mapsto 0\}\,\mathtt{x:=10}\,\{\mathtt{x} \mapsto 10\}}}{\{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0\}\,\mathtt{x:=10}\,\{\mathtt{x} \mapsto 10 * \mathtt{y} \mapsto 0\}} \qquad \dfrac{\overline{\{\mathtt{y} \mapsto 0\}\,\mathtt{y:=12}\,\{\mathtt{y} \mapsto 12\}}}{\{\mathtt{x} \mapsto 10 * \mathtt{y} \mapsto 0\}\,\mathtt{y:=12}\,\{\mathtt{x} \mapsto 10 * \mathtt{y} \mapsto 12\}}}{\{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0\}\,\mathtt{x:=10;\ y:=12}\,\{\mathtt{x} \mapsto 10 * \mathtt{y} \mapsto 12\}}$$

The *singleton assertion* $a \mapsto v$ denotes that the memory consists of exactly one object with value v at address a. The assignments are not considered in the context of the entire memory, but just in the part of the memory that is used.

The key observation that led to our separation logic for C, see also [31, 33], is the correspondence between non-determinism in expressions and a form of concurrency. Inspired by the rule for the parallel composition [46], we have rules for each operator $\circledcirc$ that are of the following shape.

$$\frac{\{P_1\}\,e_1\,\{Q_1\} \qquad \{P_2\}\,e_2\,\{Q_2\}}{\{P_1 * P_2\}\,e_1 \circledcirc e_2\,\{Q_1 * Q_2\}}$$

The intuitive idea of this rule is that if the memory can be subdivided into two parts in which the subexpressions e_1 and e_2 can be executed safely, then the expression $e_1 \circledcirc e_2$ can be executed safely in the whole memory. Non-interference of the side-effects of e_1 and e_2 is guaranteed by the separating conjunction. It ensures that the parts of the memory described by P_1 and P_2 do not have overlapping areas that will be written to. We thus effectively rule out expressions with undefined behavior such as `(x = 3) + (x = 4)` (see Section 3.6 for discussion).

Subdividing the memory into multiple parts is not a simple operation. In order to illustrate this, let us consider a shallow embedding of assertions of separation logic $P, Q : \mathsf{mem} \to \mathsf{Prop}$ (think of mem as being the set of finite partial functions from some set of object identifiers to some set of objects. The exact definition in the context of CH_2O is given in Definition 6.26). In such a shallow embedding, one would define the separating conjunction as follows:

$$P * Q := \lambda m\,.\,\exists m_1\,m_2\,.\,m = m_1 \cup m_2 \wedge P\,m_1 \wedge Q\,m_2.$$

The operation $\cup$ is *not* the disjoint union of finite partial functions, but a more fine grained operation. There are two reasons for that. Firstly, subdivision of memories should allow for partial overlap, as long as writable objects are unique to a single part. For example, the expression `x + x` has defined behavior, but the expressions `x + (x = 4)` and `(x = 3) + (x = 4)` have not.

We use separation logic with permissions [11] to deal with partial overlap of memories. That means, we equip the singleton assertion $a \overset{\gamma}{\mapsto} v$ with a permission γ. The essential property of the singleton assertion is that given a writable permission γ_w there is a readable permission γ_r with:

$$(a \overset{\gamma_w}{\longmapsto} v) \quad \leftrightarrow \quad (a \overset{\gamma_r}{\longmapsto} v) * (a \overset{\gamma_r}{\longmapsto} v).$$

The above property is an instance of a slightly more general property. We consider a binary operation $\cup$ on permissions so we can write:

$$(a \overset{\gamma_1 \cup \gamma_2}{\longmapsto} v) \quad \leftrightarrow \quad (a \overset{\gamma_1}{\longmapsto} v) * (a \overset{\gamma_2}{\longmapsto} v).$$

Secondly, it should be possible to subdivide array, struct and union objects into subobjects corresponding to their elements. For example, in the case of an array `int a[2]`, the expression `(a[0] = 1) + (a[1] = 4)` has defined behavior, and we should be able to prove so. The essential property of the singleton assertion for an $\mathsf{array}\,[\,y_0,\ldots,y_{n-1}\,]$ value is:

$$(a \overset{\gamma}{\mapsto} \mathsf{array}\,[\,v_0,\ldots,v_{n-1}\,]) \quad \leftrightarrow \quad (a[0] \overset{\gamma}{\mapsto} v_0) * \cdots * (a[n-1] \overset{\gamma}{\mapsto} v_{n-1}).$$

This paper does not describe the $\mathrm{CH_2O}$ separation logic and its shallow embedding of assertions. These are described in the author's PhD thesis [33]. Instead, we consider just the operations $\cup$ on permissions and memories.

5.2 Separation algebras

As shown in the previous section, the key operation needed to define a shallow embedding of separation logic with permissions is a binary operation $\cup$ on memories and permissions. Calcagno *et al.* introduced the notion of a *separation algebra* [13] so as to capture common properties of the $\cup$ operation. A *separation algebra* $(A, \emptyset, \cup)$ is a partial cancellative commutative monoid (see Definition 5.1 for our actual definition). Some prototypical instances of separation algebras are:

- Finite partial functions $(A \to_{\mathsf{fin}} B, \emptyset, \cup)$, where $\emptyset$ is the empty finite partial function, and $\cup$ the disjoint union on finite partial functions.
- The Booleans $(\mathsf{bool}, \mathsf{false}, \vee)$.
- Boyland's fractional permissions $([0,1]_{\mathbb{Q}}, 0, +)$ where 0 denotes no access, 1 denotes writable access, and $0 < \_ < 1$ denotes read-only access [11, 12].

Separation algebras are also closed under various constructs (such as products and finite functions), and complex instances can thus be built compositionally.

When formalizing separation algebras in the Coq proof assistant, we quickly ran into some problems:

- Dealing with partial operations such as $\cup$ is cumbersome, see Section 8.3.
- Dealing with subset types (modeled as Σ-types) is inconvenient.
- Operations such as the difference operation $\setminus$ cannot be defined constructively from the laws of a separation algebra.

In order to deal with the issue of partiality, we turn $\cup$ into a total operation. Only in case x and y are *disjoint*, notation $x \perp y$, we require $x \cup y$ to satisfy the laws of a separation algebra. Instead of using subsets, we equip separation algebras with a predicate $\mathsf{valid} : A \to \mathsf{Prop}$ that explicitly describes a subset of the carrier A. Lastly, we explicitly add a difference operation $\setminus$.

Definition 5.1 A *separation algebra* consists of a type A, with:

- An element $\emptyset : A$
- A predicate $\mathsf{valid} : A \to \mathsf{Prop}$
- Binary relations $\perp,\ \subseteq\ : A \to A \to \mathsf{Prop}$
- Binary operations $\cup,\ \setminus\ : A \to A \to A$

Satisfying the following laws:

1. If $\mathsf{valid}\ x$, then $\emptyset \perp x$ and $\emptyset \cup x = x$
2. If $x \perp y$, then $y \perp x$ and $x \cup y = y \cup x$
3. If $x \perp y$ and $x \cup y \perp z$, then $y \perp z$, $x \perp y \cup z$, and $x \cup (y \cup z) = (x \cup y) \cup z$
4. If $z \perp x$, $z \perp y$ and $z \cup x = z \cup y$, then $x = y$
5. If $x \perp y$, then $\mathsf{valid}\ x$ and $\mathsf{valid}\ (x \cup y)$
6. If $x \perp y$ and $x \cup y = \emptyset$, then $x = \emptyset$
7. If $x \perp y$, then $x \subseteq x \cup y$
8. If $x \subseteq y$, then $x \perp y \setminus x$ and $x \cup y \setminus x = y$

Laws 1–4 describe the traditional laws of a separation algebra: identity, commutativity, associativity and cancellativity. Law 5 ensures that valid is closed under the $\cup$ operation. Law 6 describes positivity. Laws 7 and 8 fully axiomatize the $\subseteq$ relation and $\setminus$ operation. Using the positivity and cancellation law, we obtain that $\subseteq$ is a partial order in which $\cup$ is order preserving and respecting.

In case of permissions, the $\emptyset$ element is used to split objects of compound types (arrays and structs) into multiple parts. We thus use separation algebras instead of *permission algebras* [47], which are a variant of separation algebras without an $\emptyset$ element.

Definition 5.2 The *Boolean separation algebra* bool is defined as:

$$\begin{aligned} \mathsf{valid}\ x &:= \mathsf{True} & \emptyset &:= \mathsf{false} \\ x \perp y &:= \neg x \vee \neg y & x \cup y &:= x \vee y \\ x \subseteq y &:= x \to y & x \setminus y &:= x \wedge \neg y \end{aligned}$$

In the case of fractional permissions $[0,1]_{\mathbb{Q}}$ the problem of partiality and subset types already clearly appears. The $\cup$ operation (here $+$) can 'overflow'. We remedy this problem by having all operations operate on pre-terms (here $\mathbb{Q}$) and the predicate valid describes validity of pre-terms (here $0 \leq \_ \leq 1$).

Definition 5.3 The *fractional separation algebra* $\mathbb{Q}$ is defined as:

$$\begin{aligned} \mathsf{valid}\ x &:= 0 \leq x \leq 1 & \emptyset &:= 0 \\ x \perp y &:= 0 \leq x, y \ \wedge\ x + y \leq 1 & x \cup y &:= x + y \\ x \subseteq y &:= 0 \leq x \leq y \leq 1 & x \setminus y &:= x - y \end{aligned}$$

The version of separation algebras by Klein *et al.* [29] in Isabelle also models $\cup$ as a total operation and uses a relation $\perp$. There are some differences:

- We include a predicate valid to prevent having to deal with subset types.
- They have weaker premises for associativity (law 3), namely $x \perp y$, $y \perp z$ and $x \perp z$ instead of $x \perp y$ and $x \cup y \perp z$. Ours are more natural, *e.g.* for fractional permissions one has $0.5 \perp 0.5$ but not $0.5 + 0.5 \perp 0.5$, and it thus makes no sense to require $0.5 \cup (0.5 \cup 0.5) = (0.5 \cup 0.5) \cup 0.5$ to hold.
- Since Coq (without axioms) does not have a choice operator, the $\setminus$ operation cannot be defined in terms of $\cup$. Isabelle has a choice operator.

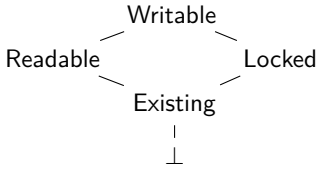
Dockins *et al.* [17] have formalized a hierarchy of different separation algebras in Coq. They have dealt with the issue of partiality by treating $\cup$ as a relation instead of a function. This is unnatural, because equational reasoning becomes impossible and one has to name all auxiliary results.

Bengtson *et al.* [6] have formalized separation algebras in Coq to reason about object-oriented programs. They have treated $\cup$ as a partial function, and have not defined any complex permission systems.

5.3 Permissions

In this section we define the CH_2O permission system and show that it forms a separation algebra. We furthermore define *permission kinds*, which are used to classify the abilities of the permissions.

Definition 5.4 The lattice of *permission kinds* $(\mathsf{pkind}, \subseteq)$ is defined as:



The order $k_1 \subseteq k_2$ expresses that k_1 has fewer abilities than k_2. This organization of permissions is inspired by that of Leroy *et al.* [40]. The intuitive meaning of the above permission kinds is as follows:

- Writable. *Writable permissions* allow reading and writing.
- Readable. *Read-only permissions* allow solely reading.
- Existing. *Existence permissions* [11] are used for objects that are known to exist but whose value cannot be used. Existence permissions are used to model that C only allows pointer arithmetic on pointers that refer to objects that have not been previously deallocated (see Section 3.4 for discussion).
- Locked. *Locked permissions* are used to formalize the sequence point restriction. When an object is modified during the execution of an expression, it is temporarily given a locked permission to forbid any read/write accesses until the next sequence point.
  For example, in `(x = 3) + *p;` the assignment `x = 3` locks the permissions of the object `x`. Since future read/write accesses to `x` are forbidden, accessing `*p` results in undefined in case `p` points to `x`. At the sequence point "`;`", the original permission of `x` is restored.
  Locked permissions are different from existence permissions because the operational semantics can change writable permissions into locked permissions and *vice versa*, but cannot do that with existence permissions.
- $\bot$. *Empty permissions* allow no operations.

In the CH_2O separation logic we do not only have control which operations are allowed, but we also have to deal with share accounting.

- We need to subdivide objects with writable or read-only permission into multiple parts with read-only permission. For example, in the expression `x + x`, both subexpressions require `x` to have at least read-only permission.
- We need to subdivide objects with writable permission into a part with existence permission and a part with writable permission. For example, in the expression `*(p + 1) = (*p = 1)`, the subexpression `*p = 1` requires `*p` to have writable permission, and the subexpression `*(p + 1)` requires `*p` to have at least existence permission in order to perform pointer arithmetic on `p`.

We combine fractional permissions with counting permissions to combine these kinds of share accounting. Counting permissions have originally been introduced by Bornat *et al.* [11].

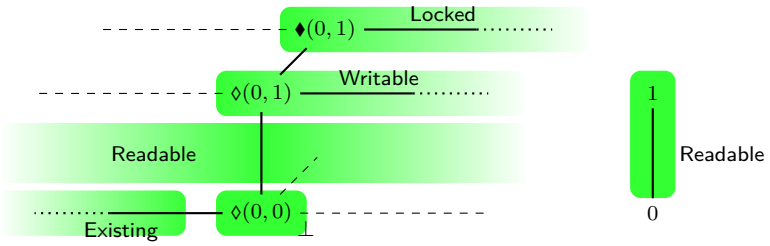
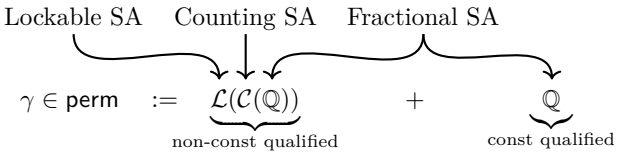


Figure 1 CH_2O permissions. The dashed lines correspond to permissions that are invalid.

Definition 5.5 *CH_2O permissions* perm are defined as:



where $\mathcal{L}(A) := \{\blacklozenge, \lozenge\} \times A$ and $\mathcal{C}(A) := \mathbb{Q} \times A$.

The author's PhD thesis [33] gives the exact definition of the separation algebra structure of permissions by defining it one by one for the counting separation algebra $\mathcal{C}$, the lockable separation algebra $\mathcal{L}$, and the separation algebra on sums $+$. We omit the formal definitions of these separation algebras in this paper.

We have three sorts of permissions:

- Unlocked permissions $\lozenge(x, y)$ where $x \in \mathbb{Q}$ counts the number existence permissions, and $y \in \mathbb{Q}$ is a fractional permission accounting for the read/write share. Permissions $\lozenge(x, 0)$ with $x < 0$ are existence permissions (see also Definitions 5.6 and 5.9). Note that the counter x is not a fractional permission and is thus not restricted to the interval $[0, 1]_{\mathbb{Q}}$.
- Locked permissions $\blacklozenge(x, y)$ where $x \in \mathbb{Q}$ counts the number existence permissions, and $y \in \mathbb{Q}$ is a fractional permission accounting for the read/write share.
- Const permissions $\gamma \in \mathbb{Q}$, which are used for `const` qualified objects. Modifying an object with const permissions results in undefined behavior. Const permissions do not have a locked variant or an existence counter as they do not allow writing.

The areas marked green in Figure 1 indicate the definition of the valid predicate on permissions. The figure furthermore visualizes how the permissions are projected onto their kinds, which is defined formally below.

Definition 5.6 The function kind : perm → pkind gives the *kind of a permission*. It is defined as follows:

$$\mathsf{kind}\ \gamma := \begin{cases} \mathsf{Locked} & \text{if } \gamma = \blacklozenge(x, y) \\ \mathsf{Writable} & \text{if } \gamma = \lozenge(x, 1) \\ \mathsf{Readable} & \text{if } \gamma = \lozenge(x, y) \text{ with } 0 < y < 1 \\ \mathsf{Existing} & \text{if } \gamma = \lozenge(x, 0) \text{ with } x \neq 0 \\ \mathsf{Readable} & \text{if } \gamma \in \mathbb{Q} \\ \bot & \text{otherwise} \end{cases}$$

Definition 5.7 The *locking operations* $\mathsf{lock}, \mathsf{unlock} : \mathsf{perm} \to \mathsf{perm}$ are defined as:

$$\begin{aligned} \mathsf{lock}\ \gamma &:= \blacklozenge(x, y) && \text{provided } \gamma = \lozenge(x, y) \\ \mathsf{unlock}\ \gamma &:= \lozenge(x, y) && \text{provided } \gamma = \blacklozenge(x, y) \end{aligned}$$

The lock operation should only be used on permissions γ with $\mathsf{Writable} \subseteq \mathsf{kind}\ \gamma$. In other cases it produces a dummy value. Likewise, unlock should only be used on permissions γ with $\mathsf{kind}\ \gamma = \mathsf{Locked}$, and produces a dummy otherwise.

The operation $\cup$ on permissions is defined as point-wise addition of the counting permission and the fractional permission, and the operation $\setminus$ is defined as point-wise subtraction. The exact definitions can be found in [33]. Apart from the common separation algebra connectives, we define an operation $\frac{1}{2}$ to subdivide a writable or read-only permission into two read-only permissions.

Definition 5.8 The *split operation* $\frac{1}{2} : \mathsf{perm} \to \mathsf{perm}$ is defined as:

$$\tfrac{1}{2}\gamma := \begin{cases} \lozenge(0.5 \cdot x, 0.5 \cdot y) & \text{if } \gamma = \lozenge(x, y) \\ 0.5 \cdot x & \text{if } \gamma = x \in \mathbb{Q} \\ \gamma & \text{otherwise, dummy value} \end{cases}$$

Given a writable or read-only permission γ, the subdivided read-only permission $\frac{1}{2}\gamma$ enjoys $\frac{1}{2}\gamma \perp \frac{1}{2}\gamma$ and $\frac{1}{2}\gamma \cup \frac{1}{2}\gamma = \gamma$. The split operation will be described abstractly by extended separation algebras in Section 5.4.

Definition 5.9 The *existence permission* token is defined as $\lozenge(-1, 0)$.

Existence permissions are used to subdivide objects with writable permission into a part with existence permission and a part with writable permission. For example, in `*(p + 1) = (*p = 1)`, the subexpression `*p = 1` requires writable permission of `*p`, and `*(p + 1)` requires an existence permission of `*p` to perform pointer arithmetic. Subdivision is achieved using the $\setminus$ operation, which can be used to split a writable permission γ into an existence permission token and writable permission $\gamma \setminus \mathsf{token}$. Law 8 of separation algebras guarantees that combining the subdivided permissions gives back the original permission, *i.e.* $\mathsf{token} \cup (\gamma \setminus \mathsf{token}) = \gamma$. Note that because tokens can be combined using $\cup$ and subdivided using $\frac{1}{2}$, the counter x in the permissions $\lozenge(x, y)$ and $\blacklozenge(x, y)$ is an arbitrary rational number.

As ensured by Definition 6.58, only objects with the full $\lozenge(0, 1)$ permission can be deallocated, whereas objects with $\gamma \setminus \mathsf{token}$ permission cannot. This is to model that expressions such as `(p == p) + (free(p),0)` have undefined behavior.

The following lemma shows that the operations on permissions interact accordingly and respect the permission kinds.

Lemma 5.10 *Permissions satisfy the following properties:*

$$\begin{aligned} \mathsf{unlock}\ (\mathsf{lock}\ \gamma) &= \gamma && \text{if } \mathsf{Writable} \subseteq \mathsf{kind}\ \gamma \\ \mathsf{kind}\ (\mathsf{lock}\ \gamma) &= \mathsf{Locked} && \text{if } \mathsf{Writable} \subseteq \mathsf{kind}\ \gamma \\ \mathsf{kind}\ \left(\tfrac{1}{2}\gamma\right) &= \begin{cases} \mathsf{Readable} & \text{if } \mathsf{Writable} \subseteq \mathsf{kind}\ \gamma \\ \mathsf{kind}\ \gamma & \text{otherwise} \end{cases} \\ \mathsf{kind}\ \mathsf{token} &= \mathsf{Existing} \\ \mathsf{kind}\ (\gamma \setminus \mathsf{token}) &= \mathsf{kind}\ \gamma \\ \mathsf{kind}\ \gamma_1 \subseteq \mathsf{kind}\ \gamma_2 && \text{if } \gamma_1 \subseteq \gamma_2 \end{aligned}$$

5.4 Extended separation algebras

We extend the notion of a separation algebra with a split operation $\frac{1}{2}$, and predicates unmapped and exclusive that associate – in an abstract way – an intended semantics to elements of a separation algebra. Recall that the split operation $\frac{1}{2}$ plays an important role in the CH_2O separation logic [31, 33] to subdivide objects with writable or read-only permission into multiple parts with read-only permission.

Definition 5.11 An *extended separation algebra* extends a separation algebra with:

- Predicates $\mathsf{splittable}, \mathsf{unmapped}, \mathsf{exclusive} : A \to \mathsf{Prop}$
- A unary operation $\frac{1}{2} : A \to A$

Satisfying the following laws:

9. If $x \perp x$, then $\mathsf{splittable}\ (x \cup x)$
10. If $\mathsf{splittable}\ x$, then $\frac{1}{2}x \perp \frac{1}{2}x$ and $\frac{1}{2}x \cup \frac{1}{2}x = x$
11. If $\mathsf{splittable}\ y$ and $x \subseteq y$, then $\mathsf{splittable}\ x$
12. If $x \perp y$ and $\mathsf{splittable}\ (x \cup y)$, then $\frac{1}{2}(x \cup y) = \frac{1}{2}x \cup \frac{1}{2}y$
13. $\mathsf{unmapped}\ \emptyset$, and if $\mathsf{unmapped}\ x$, then $\mathsf{valid}\ x$
14. If $\mathsf{unmapped}\ y$ and $x \subseteq y$, then $\mathsf{unmapped}\ x$
15. If $x \perp y$, $\mathsf{unmapped}\ x$ and $\mathsf{unmapped}\ y$, then $\mathsf{unmapped}\ (x \cup y)$
16. $\mathsf{exclusive}\ x$ iff $\mathsf{valid}\ x$ and for all y with $x \perp y$ we have $\mathsf{unmapped}\ y$
17. Not both $\mathsf{exclusive}\ x$ and $\mathsf{unmapped}\ x$
18. There exists an x with $\mathsf{valid}\ x$ and not $\mathsf{unmapped}\ x$

Note that $\frac{1}{2}$ is described by a total function whose result, $\frac{1}{2}x$, is only meaningful if $\mathsf{splittable}\ x$ holds. This is to account for locked permissions, which cannot be split. Law 11 ensures that splittable permissions are infinitely splittable, and law 12 ensures that $\frac{1}{2}$ distributes over $\cup$.

The predicates unmapped and exclusive associate an intended semantics to the elements of a separation algebra in an abstract way. The predicate unmapped describes whether the permission allows its content to be used, as will become clear in the definition of the *tagged separation algebra* (Definition 5.13). The predicate exclusive is the dual of unmapped. Let us consider the separation algebra of fractional permissions to describe the intended meaning of these predicates.

Definition 5.12 The *fractional separation algebra* $\mathbb{Q}$ is extended with:

$$\mathsf{splittable}\ x := 0 \leq x \leq 1 \qquad \frac{1}{2}x := 0.5 \cdot x$$
$$\mathsf{unmapped}\ x := x = 0 \qquad \mathsf{exclusive}\ x := x = 1$$

Remember that permissions will be used to annotate each individual bit in memory. Unmapped permissions are *on the bottom*: they do not allow their bit to be used in any way. Exclusive permissions are *on the top*: they are the sole owner of a bit and can do anything to that bit without affecting disjoint bits.

Fractional permissions have exactly one unmapped element and exactly one exclusive element, but CH_2O permissions have more structure. The elements of the CH_2O permission system are classified as follows:

unmapped	exclusive	Examples
	✓	Writable and Locked permissions
		Readable permissions
✓		The $\emptyset$ permission and Existing permissions

In order to formalize the intuitive meaning of the $\mathsf{unmapped}$ predicate and to abstractly describe bits annotated with permissions, we introduce the *tagged separation algebra* $\mathcal{T}^t_T(A)$. In the memory model it is instantiated as $\mathcal{T}^{\lightning}_{\mathsf{bit}}(\mathsf{perm})$ (Definition 6.21). The elements $(\gamma,\, b)$ consist of a permission $\gamma \in \mathsf{perm}$ and bit $b \in \mathsf{bit}$. We use the symbolic bit $\lightning$ that represents indeterminate storage to ensure that bits with $\mathsf{unmapped}$ permissions have no usable value.

Definition 5.13 Given a separation algebra A and a set of tags T with default tag $t \in T$, the *tagged separation algebra* $\mathcal{T}^t_T(A) := A \times T$ *over* A is defined as:

$$\begin{aligned}
\mathsf{valid}\ (x,\, y) &:= \mathsf{valid}\ x \land (\mathsf{unmapped}\ x \to y = t)\\
\emptyset &:= (\emptyset,\, t)\\
(x,\, y) \perp (x',\, y') &:= x \perp x' \land (\mathsf{unmapped}\ x \lor y = y' \lor \mathsf{unmapped}\ x')\\
&\qquad \land (\mathsf{unmapped}\ x \to y = t) \land (\mathsf{unmapped}\ x' \to y' = t)\\
(x,\, y) \cup (x',\, y') &:= \begin{cases} (x \cup x',\, y') & \text{if } y = t\\ (x \cup x',\, y) & \text{otherwise}\end{cases}\\
\mathsf{splittable}\ (x,\, y) &:= \mathsf{splittable}\ x \land (\mathsf{unmapped}\ x \to y = t)\\
\tfrac{1}{2}(x,\, y) &:= (\tfrac{1}{2}x,\, y)\\
\mathsf{unmapped}\ (x,\, y) &:= \mathsf{unmapped}\ x \land y = t\\
\mathsf{exclusive}\ (x,\, y) &:= \mathsf{exclusive}\ x
\end{aligned}$$

The definitions of the omitted relations and operations are as expected.

6 The memory model

This section defines the $\mathrm{CH_2O}$ memory model whose external interface consists of operations with the following types:

$$\begin{aligned}
\mathsf{lookup}_\Gamma &: \mathsf{addr} \to \mathsf{mem} \to \mathsf{option}\ \mathsf{val}\\
\mathsf{force}_\Gamma &: \mathsf{addr} \to \mathsf{mem} \to \mathsf{mem}\\
\mathsf{insert}_\Gamma &: \mathsf{addr} \to \mathsf{mem} \to \mathsf{val} \to \mathsf{mem}\\
\mathsf{writable}_\Gamma &: \mathsf{addr} \to \mathsf{mem} \to \mathsf{Prop}\\
\mathsf{lock}_\Gamma &: \mathsf{addr} \to \mathsf{mem} \to \mathsf{mem}\\
\mathsf{unlock} &: \mathsf{lockset} \to \mathsf{mem} \to \mathsf{mem}\\
\mathsf{alloc}_\Gamma &: \mathsf{index} \to \mathsf{val} \to \mathsf{bool} \to \mathsf{mem} \to \mathsf{mem}\\
\mathsf{dom} &: \mathsf{mem} \to \mathcal{P}_{\mathsf{fin}}(\mathsf{index})\\
\mathsf{freeable} &: \mathsf{addr} \to \mathsf{mem} \to \mathsf{Prop}\\
\mathsf{free} &: \mathsf{index} \to \mathsf{mem} \to \mathsf{mem}
\end{aligned}$$

Notation 6.1 We let $m\langle a\rangle_\Gamma := \mathsf{lookup}_\Gamma\ a\ m$ and $m\langle a := v\rangle_\Gamma := \mathsf{insert}_\Gamma\ a\ v\ m$.

Many of these operations depend on the typing environment Γ which assigns fields to structs and unions (Definition 4.8). This dependency is required because these operations need to be aware of the layout of structs and unions.

The operation $m\langle a\rangle_\Gamma$ yields the value stored at address a in memory m. It fails with $\bot$ if the permissions are insufficient, effective types are violated, or a is an end-of-array address. Reading from (the abstract) memory is not a pure operation. Although it does not affect the memory contents, it may affect the effective types [27, 6.5p6-7]. This happens for example in case type-punning is performed (see Section 3.3). This impurity is factored out by the operation $\mathsf{force}_\Gamma\, a\, m$.

The operation $m\langle a := v\rangle_\Gamma$ stores the value v at address a in memory m. A store is only permitted in case permissions are sufficient, effective types are not violated, and a is not an end-of-array address. The proposition $\mathsf{writable}_\Gamma\, a\, m$ describes the side-conditions necessary to perform a store.

After a successful store, the operation $\mathsf{lock}_\Gamma\, a\, m$ is used to lock the object at address a in memory m. The lock operation temporarily reduces the permissions to Locked so as to prohibit future accesses to a. Locking yields a formal treatment of the sequence point restriction (which states that modifying an object more than once between two sequence points results in undefined behavior, see Section 3.6).

The operational semantics accumulates a set $\Omega \in \mathsf{lockset}$ of addresses that have been written to (Definition 6.54) and uses the operation $\mathsf{unlock}\ \Omega\ m$ at the subsequent sequence point (which may be at the semicolon that terminates a full expression). The operation $\mathsf{unlock}\ \Omega\ m$ restores the permissions of the addresses in Ω and thereby makes future accesses to the addresses in Ω possible again. The author's PhD thesis [33] describes in detail how sequence points and locks are treated in the operational semantics.

The operation $\mathsf{alloc}_\Gamma\, o\, v\, \mu\, m$ allocates a new object with value v in memory m. The object has object identifier $o \notin \mathsf{dom}\ m$ which is non-deterministically chosen by the operation semantics. The Boolean μ expresses whether the new object is allocated by `malloc`.

Accompanying alloc_Γ, the operation $\mathsf{free}\ o\ m$ deallocates a previously allocated object with object identifier o in memory m. In order to deallocate dynamically obtained memory via `free`, the side-condition $\mathsf{freeable}\ a\ m$ describes that the permissions are sufficient for deallocation, and that a points to a `malloc`ed object.

6.1 Representation of pointers

Adapted from CompCert [40, 41], we represent memory states as finite partial functions from *object identifiers* to *objects*. Each local, global and static variable, as well as each invocation of `malloc`, is associated with a unique object identifier of a separate object in memory. This approach separates unrelated objects by construction, and is therefore well-suited for reasoning about memory transformations.

We improve on CompCert by modeling objects as structured trees instead of arrays of bytes to keep track of padding bytes and the variants of unions. This is needed to faithfully describe C11's notion of effective types (see page 4 of Section 1 for an informal description). This approach allows us to describe various undefined behaviors of C11 that have not been considered by others (see Sections 3.1 and 3.3).

In the CompCert memory model, pointers are represented as pairs (o, i) where o is an object identifier and i is a byte offset into the object with object identifier o. Since we represent objects as trees instead of as arrays of bytes, we represent pointers as paths through these trees rather than as byte offsets.

Definition 6.2 *Object identifiers* $o \in \mathsf{index}$ are elements of a fixed countable set. In the Coq development we use binary natural numbers, but since we do not rely on any properties apart from countability, we keep the representation opaque.

We first introduce a typing environment to relate the shape of paths representing pointers to the types of objects in memory.

Definition 6.3 *Memory typing environments* $\Delta \in \mathsf{memenv}$ are finite partial functions $\mathsf{index} \to_{\mathsf{fin}} (\mathsf{type} \times \mathsf{bool})$. Given a memory environment Δ:

1. An *object identifier* o *has type* τ, notation $\Delta \vdash o : \tau$, if $\Delta\, o = (\tau, \beta)$ for a β.
2. An *object identifier* o *is alive*, notation $\Delta \vdash o\ \mathsf{alive}$, if $\Delta\, o = (\tau, \mathsf{false})$ for a τ.

Memory typing environments evolve during program execution. The code below is annotated with the corresponding memory environments in red.

```
short x;
Δ1 = {o1 ↦ (signed short, false)}
int *p;
Δ2 = {o1 ↦ (signed short, false), o2 ↦ (signed int*, false)}
p = malloc(sizeof(int));
Δ3 = {o1 ↦ (signed short, false), o2 ↦ (signed int*, false), o3 ↦ (signed int, false)}
free(p);
Δ4 = {o1 ↦ (signed short, false), o2 ↦ (signed int*, false), o3 ↦ (signed int, true)}
```

Here, o_1 is the object identifier of the variable `x`, o_2 is the object identifier of the variable `p` and o_3 is the object identifier of the storage obtained via `malloc`.

Memory typing environments also keep track of objects that have been deallocated. Although one cannot directly create a pointer to a deallocated object, existing pointers to such objects remain in memory after deallocation (see the pointer `p` in the above example). These pointers, also called *dangling* pointers, cannot actually be used.

Definition 6.4 *References*, *addresses* and *pointers* are inductively defined as:

$$\begin{aligned} r \in \mathsf{refseg} &::= \xhookrightarrow{\tau[n]} i \mid \xhookrightarrow{\mathsf{struct}\ t} i \mid \xhookrightarrow{\mathsf{union}\ t}_q i \text{ with } q \in \{\circ, \bullet\} \\ \vec{r} \in \mathsf{ref} &:= \mathsf{list}\ \mathsf{refseg} \\ a \in \mathsf{addr} &::= (o : \tau, \vec{r}, i)_{\sigma >_* \sigma_\mathsf{p}} \\ p \in \mathsf{ptr} &::= \mathsf{NULL}\ \sigma_\mathsf{p} \mid a \mid f^{\vec{\tau} \mapsto \tau} \end{aligned}$$

References are paths from the top of an object in memory to some subtree of that object. The shape of references matches the structure of types:

- The reference $\xhookrightarrow{\tau[n]} i$ is used to select the ith element of a τ-array of length n.
- The reference $\xhookrightarrow{\mathsf{struct}\ t} i$ is used to select the ith field of a struct t.
- The reference $\xhookrightarrow{\mathsf{union}\ t}_q i$ is used to select the ith variant of a union t.

References can describe most pointers in C but cannot account for end-of-array pointers and pointers to individual bytes. We have therefore defined the richer notion of *addresses*. An address $(o : \tau, \vec{r}, i)_{\sigma >_* \sigma_\mathsf{p}}$ consists of:

- An object identifier o with type τ.
- A reference $\vec{r}$ to a subobject of type σ in the entire object of type τ.
- An offset i to a particular byte in the subobject of type σ (note that one cannot address individual bits in C).
- The type σ_p to which the address is cast. We use a points-to type in order to account for casts to the anonymous `void*` pointer, which is represented as the points-to type any. This information is needed to define, for example, pointer arithmetic, which is sensitive to the type of the address.

In turn, pointers extend addresses with a `NULL` pointer $\mathsf{NULL}\ \sigma_\mathsf{p}$ for each type σ_p, and function pointers $f^{\vec{\tau}\mapsto\tau}$ which contain the name and type of a function.

Let us consider the following global variable declaration:

```
struct S {
  union U { signed char x[2]; int y; } u;
  void *p;
} s;
```

The formal representation of the pointer `(void*)(s.u.x + 2)` is:

$$(o_\mathtt{s} : \mathsf{struct\ S}, \xhookrightarrow{\mathsf{struct\ S}} 0 \xhookrightarrow{\mathsf{union\ U}}_\bullet 0 \xhookrightarrow{\mathsf{signed\ char}[2]} 0, 2)_{\mathsf{signed\ char} >_* \mathsf{any}}.$$

Here, $o_\mathtt{s}$ is the object identifier associated with the variable `s` of type $\mathsf{struct\ S}$. The reference $\xhookrightarrow{\mathsf{struct\ S}} 0 \xhookrightarrow{\mathsf{union\ U}}_\bullet 0 \xhookrightarrow{\mathsf{signed\ char}[2]} 0$ and byte-offset 2 describe that the pointer refers to the third byte of the array `s.u.x`. The pointer refers to an object of type $\mathsf{signed\ char}$. The annotation any describes that the pointer has been cast to type `void*`.

The annotations $q \in \{\circ, \bullet\}$ on references $\xhookrightarrow{\mathsf{union}\ s}_q i$ describe whether type-punning is allowed or not. The annotation $\bullet$ means that type-punning is allowed, *i.e.* accessing another variant than the current one has defined behavior. The annotation $\circ$ means that type-punning is forbidden. A pointer whose annotations are all of the shape $\circ$, and thereby does not allow type-punning at all, is called *frozen*.

Definition 6.5 The *freeze* function $|\_|_\circ : \mathsf{refseg} \to \mathsf{refseg}$ is defined as:

$$|\xhookrightarrow{\tau[n]} i\,|_\circ := \xhookrightarrow{\tau[n]} i \qquad |\xhookrightarrow{\mathsf{struct}\ t} i\,|_\circ := \xhookrightarrow{\mathsf{struct}\ t} i \qquad |\xhookrightarrow{\mathsf{union}\ t}_q i\,|_\circ := \xhookrightarrow{\mathsf{union}\ t}_\circ i$$

A reference segment r is *frozen*, notation $\mathsf{frozen}\ r$, if $|\,r\,|_\circ = r$. Both $|\_|_\circ$ and frozen are lifted to references, addresses, and pointers in the expected way.

Pointers stored in memory are always in frozen shape. Definitions 6.32 and 6.41 describe the formal treatment of effective types and frozen pointers, but for now we reconsider the example from Section 3.3:

```
union U { int x; short y; } u = { .x = 3 };
short *p = &u.y;
printf("%d\n", *p);  // Undefined
printf("%d\n", u.y); // OK
```

Assuming the object `u` has object identifier $o_{\mathtt{u}}$, the pointers `&u.x`, `&u.y` and `p` have the following formal representations:

$$\begin{aligned}
\texttt{\&u.x}&: \quad (o_{\mathtt{u}} : \mathsf{union}\ \mathtt{U}, \xhookrightarrow{\mathsf{union}\ \mathtt{U}}_{\bullet} 0, 0)_{\mathsf{signed\ int} >_* \mathsf{signed\ int}}\\
\texttt{\&u.y}&: \quad (o_{\mathtt{u}} : \mathsf{union}\ \mathtt{U}, \xhookrightarrow{\mathsf{union}\ \mathtt{U}}_{\bullet} 1, 0)_{\mathsf{signed\ short} >_* \mathsf{signed\ short}}\\
\texttt{p}&: \quad (o_{\mathtt{u}} : \mathsf{union}\ \mathtt{U}, \xhookrightarrow{\mathsf{union}\ \mathtt{U}}_{\circ} 1, 0)_{\mathsf{signed\ short} >_* \mathsf{signed\ short}}
\end{aligned}$$

These pointers are likely to have the same object representation on actual computing architectures. However, due to effective types, `&u.y` may be used for type-punning but `p` may not. It is thus important that we distinguish these pointers in the formal memory model.

The additional structure of pointers is also needed to determine whether pointer subtraction has defined behavior. The behavior is only defined if the given pointers both point to an element of the same array object [27, 6.5.6p9]. Consider:

```
struct S { int a[3]; int b[3]; } s;
s.a - s.b;       // Undefined, different array objects
(s.a + 3) - s.b; // Undefined, different array objects
(s.a + 3) - s.a; // OK, same array objects
```

Here, the pointers `s.a + 3` and `s.b` have different representations in the CH_2O memory model. The author's PhD thesis [33] gives the formal definition of pointer subtraction.

We will now define typing judgments for references, addresses and pointers. The judgment for references $\Gamma \vdash \vec{r} : \tau \rightarrowtail \sigma$ states that σ is a *subobject type of* τ which can be obtained via the reference $\vec{r}$ (see also Definition 7.1). For example, `int[2]` is a subobject type of `struct S { int x[2]; int y[3]; }` via $\xhookrightarrow{\mathsf{struct}\ \mathtt{S}} 0$.

Definition 6.6 The judgment $\Gamma \vdash \vec{r} : \tau \rightarrowtail \sigma$ describes that $\vec{r}$ *is a valid reference from* τ *to* σ. It is inductively defined as:

$$\frac{}{\Gamma \vdash \varepsilon : \tau \rightarrowtail \tau} \qquad \frac{\Gamma \vdash \vec{r} : \tau \rightarrowtail \sigma[n] \qquad i < n}{\Gamma \vdash \vec{r} \xhookrightarrow{\sigma[n]} i : \tau \rightarrowtail \sigma}$$

$$\frac{\Gamma \vdash \vec{r} : \tau \rightarrowtail \mathsf{struct}\ t \quad \Gamma\, t = \vec{\sigma} \quad i < |\vec{\sigma}|}{\Gamma \vdash \vec{r} \xhookrightarrow{\mathsf{struct}\ t} i : \tau \rightarrowtail \sigma_i} \qquad \frac{\Gamma \vdash \vec{r} : \tau \rightarrowtail \mathsf{union}\ t \quad \Gamma\, t = \vec{\sigma} \quad i < |\vec{\sigma}|}{\Gamma \vdash \vec{r} \xhookrightarrow{\mathsf{union}\ t}_q i : \tau \rightarrowtail \sigma_i}$$

The typing judgment for addresses is more involved than the judgment for references. Let us first consider the following example:

```
int a[4];
```

Assuming the object `a` has object identifier $o_{\mathtt{a}}$, the end-of-array pointer `a+4` could be represented in at least the following ways (assuming $\mathsf{sizeof}\ (\mathsf{signed\ int}) = 4$):

$$\begin{aligned}
&(o_{\mathtt{a}} : \mathsf{signed\ int}[4], \xhookrightarrow{\mathsf{signed\ int}[4]} 0, 16)_{\mathsf{signed\ int} >_* \mathsf{signed\ int}}\\
&(o_{\mathtt{a}} : \mathsf{signed\ int}[4], \xhookrightarrow{\mathsf{signed\ int}[4]} 3, 4)_{\mathsf{signed\ int} >_* \mathsf{signed\ int}}
\end{aligned}$$

In order to ensure canonicity of pointer representations, we let the typing judgment for addresses ensure that the reference $\vec{r}$ of $(o : \tau, \vec{r}, i)_{\sigma >_* \sigma_\mathsf{p}}$ always refers to the first element of an array subobject. This renders the second representation illegal.

Definition 6.7 The relation $\tau >_* \sigma_\mathsf{p}$, type τ *is pointer castable to* σ_p, is inductively defined by $\tau >_* \tau$, $\tau >_* \mathsf{unsigned\ char}$, and $\tau >_* \mathsf{any}$.

Definition 6.8 The judgment $\Gamma, \Delta \vdash_* a : \sigma_\mathsf{p}$ describes that *the address* a *refers to type* σ_p. It is inductively defined as:

$$\frac{\Delta \vdash o : \tau \qquad \Gamma \vdash \tau \qquad \Gamma \vdash \vec{r} : \tau \rightarrowtail \sigma \qquad \mathsf{offset}\ \vec{r} = 0 \qquad i \leq \mathsf{sizeof}_\Gamma\ \sigma \cdot \mathsf{size}\ \vec{r} \qquad \mathsf{sizeof}_\Gamma\ \sigma_\mathsf{p} \mid i \qquad \sigma >_* \sigma_\mathsf{p}}{\Gamma, \Delta \vdash (o : \tau, \vec{r}, i)_{\sigma >_* \sigma_\mathsf{p}} : \sigma_\mathsf{p}}$$

Here, the helper functions $\mathsf{offset}, \mathsf{size} : \mathsf{ref} \to \mathbb{N}$ are defined as:

$$\mathsf{offset}\ \vec{r} := \begin{cases} i & \text{if } \vec{r} = \vec{r}_2 \xhookrightarrow{\tau[n]} i \\ 0 & \text{otherwise} \end{cases} \qquad \mathsf{size}\ \vec{r} := \begin{cases} n & \text{if } \vec{r} = \vec{r}_2 \xhookrightarrow{\tau[n]} i \\ 1 & \text{otherwise} \end{cases}$$

We use an intrinsic encoding of syntax, which means that terms contain redundant type annotations so we can read off types. Functions to read off types are named typeof and will not be defined explicitly. Type annotations make it more convenient to define operations that depend on types (such as offset and size in Definition 6.8). As usual, typing judgments ensure that type annotations are consistent.

The premises $i \leq \mathsf{sizeof}_\Gamma\ \sigma \cdot \mathsf{size}\ \vec{r}$ and $\mathsf{sizeof}_\Gamma\ \sigma_\mathsf{p} \mid i$ of the typing rule ensure that the byte offset i is aligned and within range. The inequality $i \leq \mathsf{sizeof}_\Gamma\ \sigma \cdot \mathsf{size}\ \vec{r}$ is non-strict so as to allow end-of-array pointers.

Definition 6.9 An address $a = (o : \tau, \vec{r}, i)_{\sigma >_* \sigma_\mathsf{p}}$ is called *strict*, notation $\Gamma \vdash a\ \mathsf{strict}$, in case it satisfies $i < \mathsf{sizeof}_\Gamma\ \sigma \cdot \mathsf{size}\ \vec{r}$.

The judgment $\tau >_* \sigma_\mathsf{p}$ does not describe the typing restriction of cast expressions. Instead, it defines the invariant that each address $(o : \tau, \vec{r}, i)_{\sigma >_* \sigma_\mathsf{p}}$ should satisfy. Since C is not type safe, pointer casting has $\tau >_* \sigma_\mathsf{p}$ as a run-time side-condition:

```
int x, *p = &x;
void *q = (void*)p;     // OK, signed int >_* any
int *q1 = (int*)q;      // OK, signed int >_* signed int
short *q2 = (short*)p; // Statically ill-typed
short *q3 = (short*)q; // Undefined behavior, signed int ≯_* signed short
```

Definition 6.10 The judgment $\Gamma, \Delta \vdash_* p : \sigma_\mathsf{p}$ describes that *the pointer* p *refers to type* σ_p. It is inductively defined as:

$$\frac{\Gamma \vdash_* \sigma_\mathsf{p}}{\Gamma, \Delta \vdash_* \mathsf{NULL}\ \sigma_\mathsf{p} : \sigma_\mathsf{p}} \qquad \frac{\Gamma, \Delta \vdash a : \sigma_\mathsf{p}}{\Gamma, \Delta \vdash_* a : \sigma_\mathsf{p}} \qquad \frac{\Gamma\, f = (\vec{\tau}, \tau)}{\Gamma, \Delta \vdash_* f^{\vec{\tau} \mapsto \tau} : \vec{\tau} \to \tau}$$

Addresses $(o : \tau, \vec{r} \xhookrightarrow{\sigma[n]} j, i)_{\sigma >_* \sigma_\mathsf{p}}$ that point to an element of $\tau[n]$ always have their reference point to the first element of the array, *i.e.* $j = 0$. For some operations we use the *normalized reference* which refers to the actual array element.

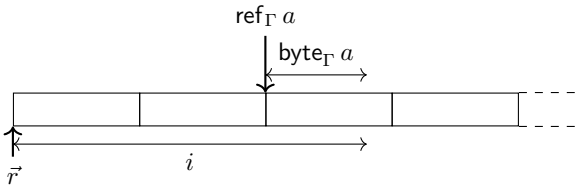
Definition 6.11 The functions $\mathsf{index} : \mathsf{addr} \to \mathsf{index}$, $\mathsf{ref}_\Gamma : \mathsf{addr} \to \mathsf{ref}$, and $\mathsf{byte}_\Gamma : \mathsf{addr} \to \mathbb{N}$ obtain the *index*, *normalized reference*, and *normalized byte offset*.

$$\begin{aligned}\mathsf{index}\ (o : \tau, \vec{r}, i)_{\sigma >_* \sigma_\mathsf{p}} &:= o\\ \mathsf{ref}_\Gamma\ (o : \tau, \vec{r}, i)_{\sigma >_* \sigma_\mathsf{p}} &:= \mathsf{setoffset}\ (i \div \mathsf{sizeof}_\Gamma\ \sigma)\ \vec{r}\\ \mathsf{byte}_\Gamma\ (o : \tau, \vec{r}, i)_{\sigma >_* \sigma_\mathsf{p}} &:= i\ \mathsf{mod}\ (\mathsf{sizeof}_\Gamma\ \sigma)\end{aligned}$$

Here, the function $\mathsf{setoffset} : \mathbb{N} \to \mathsf{ref} \to \mathsf{ref}$ is defined as:

$$\mathsf{setoffset}\ j\ \vec{r} := \begin{cases} \vec{r_2} \xhookrightarrow{\tau[n]} j & \text{if } \vec{r} = \vec{r_2} \xhookrightarrow{\tau[n]} i\\ r & \text{otherwise}\end{cases}$$

Let us display the above definition graphically. Given an address $(o : \tau, \vec{r}, i)_{\sigma >_* \sigma_\mathsf{p}}$, the normalized reference and normalized byte offset are as follows:



For end-of-array addresses the normalized reference is ill-typed because references cannot be end-of-array. For strict addresses the normalized reference is well-typed.

Definition 6.12 The judgment $\Delta \vdash p\ \mathsf{alive}$ describes that *the pointer p is alive*. It is inductively defined as:

$$\frac{}{\Delta \vdash \mathsf{NULL}\ \sigma_\mathsf{p}\ \mathsf{alive}} \qquad \frac{\Delta \vdash (\mathsf{index}\ a)\ \mathsf{alive}}{\Delta \vdash a\ \mathsf{alive}} \qquad \frac{}{\Delta \vdash f^{\vec{\tau} \mapsto \tau}\ \mathsf{alive}}$$

The judgment $\Delta \vdash o\ \mathsf{alive}$ on object identifiers is defined in Definition 6.3.

For many operations we have to distinguish addresses that refer to an entire object and addresses that refer to an individual byte of an object. We call addresses of the later kind *byte addresses*. For example:

```
int x, *p = &x;                          // p is not a byte address
unsigned char *q = (unsigned char*)&x; // q is a byte address
```

Definition 6.13 An address $(o : \tau, \vec{r}, i)_{\sigma >_* \sigma_\mathsf{p}}$ is *a byte address* if $\sigma \neq \sigma_\mathsf{p}$.

To express that memory operations commute (see for example Lemma 6.36), we need to express that addresses are *disjoint*, meaning they do not overlap. Addresses do not overlap if they belong to different objects or take a different branch at an array or struct. Let us consider an example:

```
union { struct { int x, y; } s; int z; } u1, u2;
```

The pointers `&u1` and `&u2` are disjoint because they point to separate memory objects. Writing to one does not affect the value of the other and *vice versa*. Likewise, `&u1.s.x` and `&u1.s.y` are disjoint because they point to different fields of the same struct, and as such do not affect each other. The pointers `&u1.s.x` and `&u1.z` are not disjoint because they point to overlapping objects and thus do affect each other.

Definition 6.14 *Disjointness of references* $\vec{r}_1$ *and* $\vec{r}_2$, notation $\vec{r}_1 \perp \vec{r}_2$, is inductively defined as:

$$\frac{|\,\vec{r}_1\,|_\circ = |\,\vec{r}_2\,|_\circ \qquad i \neq j}{\vec{r}_1 \xhookrightarrow{\sigma[n]} i\,\vec{r}_3 \perp \vec{r}_2 \xhookrightarrow{\sigma[n]} j\,\vec{r}_4} \qquad \frac{|\,\vec{r}_1\,|_\circ = |\,\vec{r}_2\,|_\circ \qquad i \neq j}{\vec{r}_1 \xhookrightarrow{\mathsf{struct}\ t} i\,\vec{r}_3 \perp \vec{r}_2 \xhookrightarrow{\mathsf{struct}\ t} j\,\vec{r}_4}$$

Note that we do not require a special case for $|\,\vec{r}_1\,|_\circ \neq |\,\vec{r}_2\,|_\circ$. Such a case is implicit because disjointness is defined in terms of prefixes.

Definition 6.15 *Disjointness of addresses* a_1 *and* a_2, notation $a_1 \perp_\Gamma a_2$, is inductively defined as:

$$\frac{\mathsf{index}\ a_1 \neq \mathsf{index}\ a_2}{a_1 \perp_\Gamma a_2} \qquad \frac{\mathsf{index}\ a_1 = \mathsf{index}\ a_2 \qquad \mathsf{ref}_\Gamma\ a_1 \perp \mathsf{ref}_\Gamma\ a_2}{a_1 \perp_\Gamma a_2}$$

$$\text{both } a_1 \text{ and } a_2 \text{ are byte addresses}$$

$$\frac{\mathsf{index}\ a_1 = \mathsf{index}\ a_2 \qquad |\,\mathsf{ref}_\Gamma\ a_1\,|_\circ = |\,\mathsf{ref}_\Gamma\ a_2\,|_\circ \qquad \mathsf{byte}_\Gamma\ a_1 \neq \mathsf{byte}_\Gamma\ a_2}{a_1 \perp_\Gamma a_2}$$

The first inference rule accounts for addresses whose object identifiers are different, the second rule accounts for addresses whose references are disjoint, and the third rule accounts for addresses that point to different bytes of the same subobject.

Definition 6.16 The *reference bit-offset* $\mathsf{bitoffset}_\Gamma : \mathsf{refseg} \to \mathbb{N}$ is defined as:

$$\begin{aligned}
\mathsf{bitoffset}_\Gamma\ (\xhookrightarrow{\tau[n]} i) &:= i \cdot \mathsf{bitsizeof}_\Gamma\ \tau \\
\mathsf{bitoffset}_\Gamma\ (\xhookrightarrow{\mathsf{union}\ t}_q i) &:= 0 \\
\mathsf{bitoffset}_\Gamma\ (\xhookrightarrow{\mathsf{struct}\ t} i) &:= \mathsf{bitoffsetof}_\Gamma\ \vec{\tau}\ i \quad \text{where } \Gamma\,t = \vec{\tau}
\end{aligned}$$

Moreover, we let $\mathsf{bitoffset}_\Gamma\ a := \Sigma_i\,(\mathsf{bitoffset}_\Gamma\ (\mathsf{ref}_\Gamma\ a)_i) + \mathsf{byte}_\Gamma\ a \cdot \mathsf{char\_bits}$.

Disjointness implies non-overlapping bit-offsets, but the reverse implication does not always hold because references to different variants of unions are not disjoint. For example, given the declaration `union { struct { int x, y; } s; int z; } u`, the pointers corresponding to `&u.s.y` and `&u.z` are not disjoint.

Lemma 6.17 *If* $\Gamma, \Delta \vdash a_1 : \sigma_1$, $\Gamma, \Delta \vdash a_2 : \sigma_2$, $\Gamma \vdash \{a_1, a_2\}\ \mathsf{strict}$, $a_1 \perp_\Gamma a_2$, *and* $\mathsf{index}\ a_1 \neq \mathsf{index}\ a_2$, *then either:*

1. $\mathsf{bitoffset}_\Gamma\ a_1 + \mathsf{bitsizeof}_\Gamma\ \sigma_1 \leq \mathsf{bitoffset}_\Gamma\ a_2$, *or*
2. $\mathsf{bitoffset}_\Gamma\ a_2 + \mathsf{bitsizeof}_\Gamma\ \sigma_2 \leq \mathsf{bitoffset}_\Gamma\ a_1$.

Lemma 6.18 (Well-typed addresses are properly aligned) *If* $\Gamma, \Delta \vdash a : \sigma$, *then* $(\mathsf{alignof}_\Gamma\ \sigma \cdot \mathsf{char\_bits}) \mid \mathsf{bitoffset}_\Gamma\ a$.

6.2 Representation of bits

As shown in Section 3.1, each object in C can be interpreted as an **`unsigned char`** array called the *object representation*. On actual computing architectures, the object representation consists of a sequence of concrete bits (zeros and ones). However, so as to accurately describe all undefined behaviors, we need a special treatment for the object representations of pointers and indeterminate memory in the formal memory model. To that end, $\mathrm{CH_2O}$ represents the bits belonging to the object representations of pointers and indeterminate memory symbolically.

Definition 6.19 *Bits* are inductively defined as:

$$b \in \mathsf{bit} ::= \lightning \mid 0 \mid 1 \mid (\mathsf{ptr}\ p)_i.$$

Definition 6.20 The judgment $\Gamma, \Delta \vdash b$ describes that a bit b is *valid*. It is inductively defined as:

$$\frac{}{\Gamma, \Delta \vdash \lightning} \qquad \frac{\beta \in \{0, 1\}}{\Gamma, \Delta \vdash \beta} \qquad \frac{\Gamma, \Delta \vdash_* p : \sigma_\mathsf{p} \qquad \mathsf{frozen}\ p \qquad i < \mathsf{bitsizeof}_\Gamma\,(\sigma_\mathsf{p}*)}{\Gamma, \Delta \vdash (\mathsf{ptr}\ p)_i}$$

A bit is either a concrete bit 0 or 1, the ith fragment bit $(\mathsf{ptr}\ p)_i$ of a pointer p, or the indeterminate bit $\lightning$. Integers are represented using concrete sequences of bits, and pointers as sequences of fragment bits. Assuming $\mathsf{bitsizeof}\ (\mathsf{signed\ int}*) = 32$, a pointer p to a $\mathsf{signed\ int}$ will be represented as the bit sequence $(\mathsf{ptr}\ p)_0 \ldots (\mathsf{ptr}\ p)_{31}$, and assuming $\mathsf{bitsizeof}\ (\mathsf{signed\ int}) = 32$ on a little-endian architecture, the integer $33 : \mathsf{signed\ int}$ will be represented as the bit sequence $\mathtt{1000010000000000}$.

The approach using a combination of symbolic and concrete bits is similar to Leroy *et al.* [40] and has the following advantages:

- Symbolic bit representations for pointers avoid the need to clutter the memory model with subtle, implementation-defined, and run-time dependent operations to decode and encode pointers as concrete bit sequences.
- We can precisely keep track of memory areas that are uninitialized. Since these memory areas consist of arbitrary concrete bit sequences on actual machines, most operations on them have undefined behavior.
- While reasoning about program transformations one has to relate the memory states during the execution of the source program to those during the execution of the target program. Program transformations can, among other things, make more memory defined (that is, transform some indeterminate $\lightning$ bits into determinate bits) and relabel the memory. Symbolic bit representations make it easy to deal with such transformations (see Section 7.2).
- It vastly decreases the amount of non-determinism, making it possible to evaluate the memory model as part of an executable semantics [33, 37].
- The use of concrete bit representations for integers still gives a semantics to many low-level operations on integer representations.

A small difference with Leroy *et al.* [40] is that the granularity of our memory model is on the level of bits rather than bytes. Currently we do not make explicit use of this granularity, but it allows us to support bit-fields more faithfully with respect to the C11 standard in future work.

Objects in our memory model are annotated with permissions. We use permission annotations on the level of individual bits, rather than on the level of bytes or entire objects, to obtain the most precise way of permission handling.

Definition 6.21 *Permission annotated bits* are defined as:

$$\mathbf{b} \in \mathsf{pbit} := \mathcal{T}^{\lightning}_{\mathsf{bit}}(\mathsf{perm}) = \mathsf{perm} \times \mathsf{bit}.$$

In the above definition, $\mathcal{T}$ is the tagged separation algebra that has been defined in Definition 5.13. We have spelled out its definition for brevity's sake.

Definition 6.22 The judgment $\Gamma, \Delta \vdash \mathbf{b}$ describes that a permission annotated bit $\mathbf{b}$ is *valid*. It is inductively defined as:

$$\frac{\Gamma, \Delta \vdash b \qquad \mathsf{valid}\ \gamma \qquad b = \lightning \text{ in case } \mathsf{unmapped}\ \gamma}{\Gamma, \Delta \vdash (\gamma, b)}$$

6.3 Representation of the memory

Memory trees are abstract trees whose structure corresponds to the shape of data types in C. They are used to describe individual objects (base values, arrays, structs, and unions) in memory. The memory is a forest of memory trees.

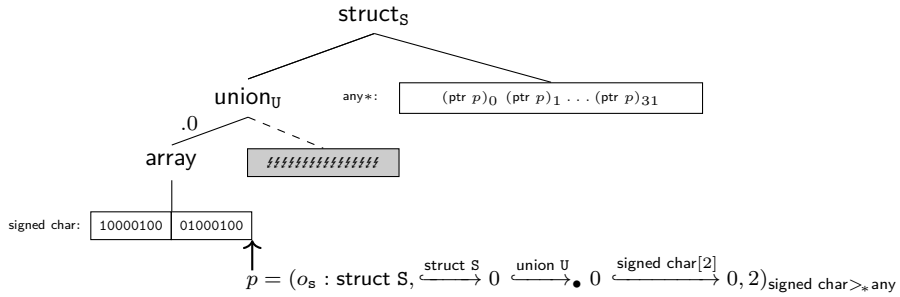
Definition 6.23 *Memory trees* are inductively defined as:

$$w \in \mathsf{mtree} ::= \mathsf{base}_{\tau_\mathsf{b}}\ \vec{\mathbf{b}} \mid \mathsf{array}_\tau\ \vec{w} \mid \mathsf{struct}_t\ \overrightarrow{w\mathbf{b}} \mid \mathsf{union}_t\ (i, w, \vec{\mathbf{b}}) \mid \overline{\mathsf{union}}_t\ \vec{\mathbf{b}}.$$

The structure of memory trees is close to the structure of types (Definition 4.7) and thus reflects the expected semantics of types: arrays are lists, structs are tuples, and unions are sums. Let us consider the following example:

```
struct S {
  union U { signed char x[2]; int y; } u; void *p;
} s = { .u = { .x = {33,34} }, .p = s.u.x + 2 };
```

The memory tree representing the object `s` with object identifier $o_\mathtt{s}$ may be as follows (permissions are omitted for brevity's sake, and integer encoding and padding are subject to implementation-defined behavior):



The representation of unions requires some explanation. We considered two kinds of memory trees for unions:

– The memory tree $\mathsf{union}_t\ (i, w, \vec{\mathbf{b}})$ represents a union whose variant is i. Unions of variant i can only be accessed through a pointer to variant i. This is essential for effective types. The list $\vec{\mathbf{b}}$ represents the padding after the element w.

– The memory tree $\overline{\mathsf{union}_t}\,\vec{\mathbf{b}}$ represents a union whose variant is yet unspecified. Whenever the union is accessed through a pointer to variant i, the list $\vec{\mathbf{b}}$ will be interpreted as a memory tree of the type belonging to the ith variant.

The reason that we consider unions $\overline{\mathsf{union}_t}\,\vec{\mathbf{b}}$ with unspecific variant at all is that in some cases the variant cannot be known. Unions that have not been initialized do not have a variant yet. Also, when a union object is constructed byte-wise through its object representation, the variant cannot be known.

Although unions are tagged in the formal memory, actual compilers implement untagged unions. Information about variants should thus be internal to the formal memory model. In Section 7.2 we prove that this is indeed the case.

The additional structure of memory trees, namely type annotations, variants of unions, and structured information about padding, can be erased by flattening. Flattening just appends the bytes on the leaves of the tree.

Definition 6.24 The *flatten operation* $\overline{(\_)} : \mathsf{mtree} \to \mathsf{list}\ \mathsf{pbit}$ is defined as:

$$\overline{\mathsf{base}_{\tau_\mathsf{b}}\,\vec{\mathbf{b}}} := \vec{\mathbf{b}} \qquad \overline{\mathsf{array}_\tau\,\vec{w}} := \overline{w_0} \ldots \overline{w_{|\vec{w}|-1}}$$

$$\overline{\mathsf{struct}_t\,\overrightarrow{w\vec{\mathbf{b}}}} := (\overline{w_0}\,\vec{\mathbf{b}}_0) \ldots (\overline{w_{|\vec{w}|-1}}\,\vec{\mathbf{b}}_{|\vec{w}|-1}) \qquad \overline{\mathsf{union}_t\,(j, w, \vec{\mathbf{b}})} := \overline{w}\,\vec{\mathbf{b}} \qquad \overline{\overline{\mathsf{union}_t}\,\vec{\mathbf{b}}} := \vec{\mathbf{b}}$$

The flattened version of the memory tree representing the object $\mathtt{s}$ in the previous example is as follows:

$$10000100\ 01000100\ \lightning\lightning\lightning\lightning\lightning\lightning\lightning\lightning\ \lightning\lightning\lightning\lightning\lightning\lightning\lightning\lightning\ (\mathsf{ptr}\ p)_0\ (\mathsf{ptr}\ p)_1 \ldots (\mathsf{ptr}\ p)_{31}$$

Definition 6.25 The judgment $\Gamma, \Delta \vdash w : \tau$ describes that the *memory tree* w *has type* τ. It is inductively defined as:

$$\frac{\Gamma \vdash_\mathsf{b} \tau_\mathsf{b} \qquad \Gamma, \Delta \vdash \vec{\mathbf{b}} \qquad |\vec{\mathbf{b}}| = \mathsf{bitsizeof}_\Gamma\,\tau_\mathsf{b}}{\Gamma, \Delta \vdash \mathsf{base}_{\tau_\mathsf{b}}\,\vec{\mathbf{b}} : \tau_\mathsf{b}} \qquad \frac{\Gamma, \Delta \vdash \vec{w} : \tau \qquad |\vec{w}| = n \neq 0}{\Gamma, \Delta \vdash \mathsf{array}_\tau\,\vec{w} : \tau[n]}$$

$$\frac{\begin{array}{c}\Gamma\,t = \vec{\tau} \qquad \Gamma, \Delta \vdash \vec{w} : \vec{\tau} \\ \forall i\,.\,(\Gamma, \Delta \vdash \vec{\mathbf{b}}_i \qquad \vec{\mathbf{b}}_i\ \text{all}\ \lightning \qquad |\vec{\mathbf{b}}_i| = (\mathsf{fieldbitsizes}_\Gamma\ \vec{\tau})_i - \mathsf{bitsizeof}_\Gamma\,\tau_i)\end{array}}{\Gamma, \Delta \vdash \mathsf{struct}_t\,\overrightarrow{w\vec{\mathbf{b}}} : \mathsf{struct}\ t}$$

$$\frac{\begin{array}{c}\Gamma\,t = \vec{\tau} \qquad i < |\vec{\tau}| \qquad \Gamma, \Delta \vdash w : \tau_i \qquad \Gamma, \Delta \vdash \vec{\mathbf{b}} \qquad \vec{\mathbf{b}}\ \text{all}\ \lightning \\ \mathsf{bitsizeof}_\Gamma\,(\mathsf{union}\ t) = \mathsf{bitsizeof}_\Gamma\,\tau_i + |\vec{\mathbf{b}}| \qquad \neg\mathsf{unmapped}\ (\overline{w}\,\vec{\mathbf{b}})\end{array}}{\Gamma, \Delta \vdash \mathsf{union}_t\,(i, w, \vec{\mathbf{b}}) : \mathsf{union}\ t}$$

$$\frac{\Gamma\,t = \vec{\tau} \qquad \Gamma, \Delta \vdash \vec{\mathbf{b}} \qquad |\vec{\mathbf{b}}| = \mathsf{bitsizeof}_\Gamma\,(\mathsf{union}\ t)}{\Gamma, \Delta \vdash \overline{\mathsf{union}_t}\,\vec{\mathbf{b}} : \mathsf{union}\ t}$$

Although padding bits should be kept indeterminate (see Section 3.1), padding bits are explicitly stored in memory trees for uniformity's sake. The typing judgment ensures that the value of each padding bit is $\lightning$ and that the padding thus only have a permission. Storing a value in padding is a no-op (see Definition 6.35).

The side-condition $\neg\mathsf{unmapped}\ (\overline{w}\,\vec{\mathbf{b}})$ in the typing rule for a union $\mathsf{union}_t\,(i, w, \vec{\mathbf{b}})$ of a specified variant ensures canonicity. Unions whose permissions are unmapped cannot be accessed and should therefore be in an unspecified variant. This condition is essential for the separation algebra structure, see Section 7.4.

Definition 6.26 *Memories* are defined as:

$$m \in \mathsf{mem} := \mathsf{index} \to_{\mathsf{fin}} (\mathsf{mtree} \times \mathsf{bool} + \mathsf{type}).$$

Each object (w, μ) in memory is annotated with a Boolean μ to describe whether it has been allocated using `malloc` (in case $\mu = \mathsf{true}$) or as a block scope local, static, or global variable (in case $\mu = \mathsf{false}$). The types of deallocated objects are kept to ensure that dangling pointers (which may remain to exist in memory, but cannot be used) have a unique type.

Definition 6.27 The judgment $\Gamma, \Delta \vdash m$ describes that *the memory m is valid*. It is defined as the conjunction of:

1. For each o and τ with $m\,o = \tau$ we have:
   (a) $\Delta \vdash o : \tau$, (b) $\Delta \nvdash o\ \mathsf{alive}$, and (c) $\Gamma \vdash \tau$.
2. For each o, w and μ with $m\,o = (w, \mu)$ we have:
   (a) $\Delta \vdash o : \tau$, (b) $\Delta \vdash o\ \mathsf{alive}$, (c) $\Gamma, \Delta \vdash w : \tau$, and (d) not $\overline{w}$ all $(\emptyset, \lightning)$.

The judgment $\Delta \vdash o\ \mathsf{alive}$ on object identifiers is defined in Definition 6.3.

Definition 6.28 The *minimal memory typing environment* $\overline{m} \in \mathsf{memenv}$ of a memory m is defined as:

$$\overline{m} := \lambda o\,.\ \begin{cases} (\tau, \mathsf{true}) & \text{if } m\,o = \tau \\ (\mathsf{typeof}\ w, \mathsf{false}) & \text{if } m\,o = (w, \mu) \end{cases}$$

Notation 6.29 We let $\Gamma \vdash m$ denote $\Gamma, \overline{m} \vdash m$.

Many of the conditions of the judgment $\Gamma, \Delta \vdash m$ ensure that the types of m match up with the types in the memory environment Δ (see Definition 6.3). One may of course wonder why do we not define the judgment $\Gamma \vdash m$ directly, and even consider typing of a memory in an arbitrary memory environment. Consider:

```
int x = 10, *p = &x;
```

Using an assertion of separation logic we can describe the memory induced by the above program as $\mathtt{x} \mapsto 10 * \mathtt{p} \mapsto \&\mathtt{x}$. The separation conjunction $*$ describes that the memory can be subdivided into two parts, a part for `x` and another part for `p`. When considering $\mathtt{p} \mapsto \&\mathtt{x}$ in isolation, which is common in separation logic, we have a pointer that refers outside the part itself. This isolated part is thus not typeable by $\Gamma \vdash m$, but it is typeable in the context of a the memory environment corresponding to the whole memory. See also Lemma 7.26.

In the remaining part of this section we will define various auxiliary operations that will be used to define the memory operations in Section 6.5. We give a summary of the most important auxiliary operations:

$$\begin{aligned} \mathsf{new}^{\gamma}_{\Gamma} &: \mathsf{type} \to \mathsf{mtree} && \text{for} \quad \gamma : \mathsf{perm} \\ (\_)[\,\_\,]_{\Gamma} &: \mathsf{mem} \to \mathsf{addr} \to \mathsf{option\ mtree} \\ (\_)[\_/f]_{\Gamma} &: \mathsf{mem} \to \mathsf{addr} \to \mathsf{mem} && \text{for} \quad f : \mathsf{mtree} \to \mathsf{mtree} \end{aligned}$$

Intuitively these are just basic tree operations, but unions make their actual definitions more complicated. The indeterminate memory tree $\mathsf{new}^{\gamma}_{\Gamma}\,\tau$ consists of

indeterminate bits with permission γ, the lookup operation $m[a]_\Gamma$ yields the memory tree at address a in m, and the alter operation $m[a/f]_\Gamma$ applies the function f to the memory tree at address a in m.

The main delicacy of all of these operations is that we sometimes have to interpret bits as memory trees, or reinterpret memory trees as memory trees of a different type. Most notably, reinterpretation is needed when type-punning is performed:

```
union int_or_short { int x; short y; } u = { .x = 3 };
short z = u.y;
```

This code will reinterpret the bit representation of a memory tree representing an `int` value `3` as a memory tree of type `short`. Likewise:

```
union int_or_short { int x; short y; } u;
((unsigned char*)&u)[0] = 3;
((unsigned char*)&u)[1] = 0;
short z = u.y;
```

Here, we poke some bytes into the object representation of `u`, and interpret these as a memory tree of type `short`.

We have defined the flatten operation $\overline{w}$ that takes a memory tree w and yields its bit representation already in Definition 6.24. We now define the operation which goes in opposite direction, called the *unflatten operation*.

Definition 6.30 The *unflatten operation* $(\_)^\tau_\Gamma : \mathsf{list\ pbit} \to \mathsf{mtree}$ is defined as:

$$
\begin{aligned}
(\vec{\mathbf{b}})^{\tau_\mathsf{b}}_\Gamma &:= \mathsf{base}_{\tau_\mathsf{b}}\ \vec{\mathbf{b}} \\
(\vec{\mathbf{b}})^{\tau[n]}_\Gamma &:= \mathsf{array}_\tau\ ((\vec{\mathbf{b}}_{[0,\,s)})^\tau_\Gamma \ldots (\vec{\mathbf{b}}_{[(n-1)s,\,ns)})^\tau_\Gamma) \text{ where } s := \mathsf{bitsizeof}_\Gamma\,\tau \\
(\vec{\mathbf{b}})^{\mathsf{struct}\ t}_\Gamma &:= \mathsf{struct}_t \begin{pmatrix} (\vec{\mathbf{b}}_{[0,\,s_0)})^{\tau_0}_\Gamma\ \vec{\mathbf{b}}^{\lightning}_{[s_0,\,z_1)} \\ \ldots \\ (\vec{\mathbf{b}}_{[z_{n-1},\,z_{n-1}+s_{n-1})})^{\tau_{n-1}}_\Gamma\ \vec{\mathbf{b}}^{\lightning}_{[z_{n-1}+s_{n-1},\,z_n)} \end{pmatrix} \\
&\qquad \text{where } \Gamma\,t = \vec{\tau},\ n := |\vec{\tau}|,\ s_i := \mathsf{bitsizeof}_\Gamma\,\tau_i \text{ and } z_i := \mathsf{bitoffsetof}_\Gamma\,\vec{\tau}\ i \\
(\vec{\mathbf{b}})^{\mathsf{union}\ t}_\Gamma &:= \overline{\mathsf{union}_t}\ \vec{\mathbf{b}}
\end{aligned}
$$

Here, the operation $(\_)^{\lightning} : \mathsf{pbit} \to \mathsf{pbit}$ is defined as $(x,\, b)^{\lightning} := (x,\, \lightning)$.

Now, the need for $\overline{\mathsf{union}_t}\ \vec{\mathbf{b}}$ memory trees becomes clear. While unflattening a bit sequence as a union, there is no way of knowing which variant of the union the bits constitute. The operations $\overline{(\_)}$ and $(\_)^\tau_\Gamma$ are neither left nor right inverses:

- We do not have $(\overline{w})^\tau_\Gamma = w$ for each w with $\Gamma, \Delta \vdash w : \tau$. Variants of unions are destroyed by flattening w.
- We do not have $\overline{(\vec{\mathbf{b}})^\tau_\Gamma} = \vec{\mathbf{b}}$ for each $\vec{\mathbf{b}}$ with $|\vec{\mathbf{b}}| = \mathsf{bitsizeof}_\Gamma\,\tau$ either. Padding bits become indeterminate due to $(\_)^{\lightning}$ by unflattening.

In Section 7.2 we prove weaker variants of these cancellation properties that are sufficient for proofs about program transformations.

Definition 6.31 Given a permission $\gamma \in \mathsf{perm}$, the operation $\mathsf{new}^{\gamma}_{\Gamma} : \mathsf{type} \to \mathsf{mtree}$ that yields the indeterminate memory tree is defined as:

$$\mathsf{new}^{\gamma}_{\Gamma}\,\tau := ((\gamma,\, \lightning)^{\mathsf{bitsizeof}_\Gamma\,\tau})^{\tau}_{\Gamma}.$$

The memory tree $\mathsf{new}^{\gamma}_{\Gamma}\,\tau$ that consists of indeterminate bits with permission γ is used for objects with indeterminate value. We have defined $\mathsf{new}^{\gamma}_{\Gamma}\,\tau$ in terms of the unflattening operation for simplicity's sake. This definition enjoys desirable structural properties such as $\mathsf{new}^{\gamma}_{\Gamma}\,(\tau[n]) = (\mathsf{new}^{\gamma}_{\Gamma}\,\tau)^n$.

We will now define the lookup operation $m[a]_\Gamma$ that yields the subtree at address a in the memory m. The lookup function is partial, it will fail in case a is end-of-array or violates effective types. We first define the counterpart of lookup on memory trees and then lift it to memories.

Definition 6.32 The *lookup operation on memory trees* $(\_)[\,\_\,]_\Gamma : \mathsf{mtree} \to \mathsf{ref} \to \mathsf{option\ mtree}$ is defined as:

$$\begin{aligned}
w[\varepsilon]_\Gamma &:= w\\
(\mathsf{array}_\tau\ \vec{w})[(\xhookrightarrow{\tau[n]}\ i)\ \vec{r}]_\Gamma &:= w_i[\vec{r}]_\Gamma\\
(\mathsf{struct}_t\ \overrightarrow{w\vec{\mathbf{b}}})[(\xhookrightarrow{\mathsf{struct}\ t}\ i)\ \vec{r}]_\Gamma &:= w_i[\vec{r}]_\Gamma\\
(\mathsf{union}_t\ (j, w, \vec{\mathbf{b}}))[(\xhookrightarrow{\mathsf{union}\ t}_q\ i)\ \vec{r}]_\Gamma &:= \begin{cases} w[\vec{r}]_\Gamma & \text{if } i = j\\ ((\overline{w}\,\vec{\mathbf{b}})_{[0,\,s)})^{\tau_i}_\Gamma[\vec{r}]_\Gamma & \text{if } i \neq j,\ q = \bullet,\ \mathsf{exclusive}\ (\overline{w}\,\vec{\mathbf{b}})\\ \bot & \text{if } i \neq j,\ q = \circ \end{cases}\\
&\qquad \text{where } \Gamma\,t = \vec{\tau} \text{ and } s = \mathsf{bitsizeof}_\Gamma\,\tau_i\\
(\overline{\mathsf{union}_t}\ \vec{\mathbf{b}})[(\xhookrightarrow{\mathsf{union}\ t}_q\ i)\ \vec{r}]_\Gamma &:= (\vec{\mathbf{b}}_{[0,\,\mathsf{bitsizeof}_\Gamma\,\tau_i)})^{\tau_i}_\Gamma[\vec{r}]_\Gamma \quad \text{if } \Gamma\,t = \vec{\tau},\ \mathsf{exclusive}\ \vec{\mathbf{b}}
\end{aligned}$$

The lookup operation uses the annotations $q \in \{\circ, \bullet\}$ on $\xhookrightarrow{\mathsf{union}\ s}_q\ i$ to give a formal semantics to the *strict-aliasing restrictions* [27, 6.5.2.3].

- The annotation $q = \bullet$ allows a union to be accessed via a reference whose variant is unequal to the current one. This is called type-punning.
- The annotation $q = \circ$ allows a union to be accessed only via a reference whose variant is equal to the current one. This means, it rules out type-punning.

Failure of type-punning is captured by partiality of the lookup operation. The behavior of type-punning of $\mathsf{union}_t\ (j, w, \vec{\mathbf{b}})$ via a reference to variant i is described by the conversion $((\overline{w}\,\vec{\mathbf{b}})_{[0,\,\mathsf{bitsizeof}_\Gamma\,\tau_i)})^{\tau_i}_\Gamma$. The memory tree w is converted into bits and reinterpreted as a memory tree of type τ_i.

Definition 6.33 The *lookup operation on memories* $(\_)[\,\_\,]_\Gamma : \mathsf{mem} \to \mathsf{addr} \to \mathsf{option\ mtree}$ is defined as:

$$m[a]_\Gamma := \begin{cases} ((\overline{w[\mathsf{ref}_\Gamma\,a]_\Gamma})_{[i,\,j)})^{\mathsf{unsigned\ char}}_\Gamma & \text{if } a \text{ is a byte address}\\ w[\mathsf{ref}_\Gamma\,a]_\Gamma & \text{if } a \text{ is not a byte address} \end{cases}$$

provided that $m\,(\mathsf{index}\ a) = (w,\, \mu)$. In omitted cases the result is $\bot$. In this definition we let $i := \mathsf{byte}_\Gamma\,a \cdot \mathsf{char\_bits}$ and $j := (\mathsf{byte}_\Gamma\,a + 1) \cdot \mathsf{char\_bits}$.

We have to take special care of addresses that refer to individual bytes rather than whole objects. Consider:

```
struct S { int x; int y; } s = { .x = 1, .y = 2 };
unsigned char z = ((unsigned char*)&s)[0];
```

In this code, we obtain the first byte `((unsigned char*)&s)[0]` of the struct `s`. This is formalized by flattening the entire memory tree of the struct `s`, and selecting the appropriate byte.

The C11 standard's description of effective types [27, 6.5p6-7] states that an access (which is either a read or store) affects the effective type of the accessed object. This means that although reading from memory does not affect the memory contents, it may still affect the effective types. Let us consider an example where it is indeed the case that effective types are affected by a read:

```
short g(int *p, short *q) {
  short z = *q; *p = 10; return z;
}
int main() {
  union int_or_short { int x; short y; } u;
  // initialize u with zeros, the variant of u remains unspecified
  for (size_t i = 0; i < sizeof(u); i++) ((unsigned char*)&u)[i] = 0;
  return g(&u.x, &u.y);
}
```

In this code, the variant of the union `u` is initially unspecified. The read `*q` in `g` *forces* its variant to `y`, making the assignment `*p` to variant `x` undefined. Note that it is important that we also assign undefined behavior to this example, a compiler may assume `p` and `q` to not alias regardless of how `g` is called.

We factor these side-effects out using a function $\mathsf{force}_\Gamma : \mathsf{addr} \to \mathsf{mem} \to \mathsf{mem}$ that updates the effective types (that is the variants of unions) after a successful lookup. The force_Γ function, as defined in Definition 6.5, can be described in terms of the alter operation $m[a/f]_\Gamma$ that applies the function $f : \mathsf{mtree} \to \mathsf{mtree}$ to the object at address a in the memory m and update variants of unions accordingly to a. To define force_Γ we let f be the identify.

Definition 6.34 Given a function $f : \mathsf{mtree} \to \mathsf{mtree}$, the *alter operation on memory trees* $(\_)[\_/f]_\Gamma : \mathsf{mtree} \to \mathsf{ref} \to \mathsf{mtree}$ is defined as:

$$w[\varepsilon/f]_\Gamma := f\,w$$

$$(\mathsf{array}_\tau\,\vec{w})[(\xrightarrow{\tau[n]} i)\,\vec{r}/f]_\Gamma := \mathsf{array}_\tau\,(\vec{w}[i := w_i[\vec{r}/f]_\Gamma])$$

$$(\mathsf{struct}_t\,\overrightarrow{w\vec{\mathbf{b}}})[(\xrightarrow{\mathsf{struct}\ t} i)\,\vec{r}/f]_\Gamma := \mathsf{struct}_t\,(\overrightarrow{(w\vec{\mathbf{b}})[i := w_i[\vec{r}/f]_\Gamma\vec{\mathbf{b}}_i]})$$

$$(\mathsf{union}_t\,(i, w, \vec{\mathbf{b}}))[(\xrightarrow{\mathsf{union}\ t}_q j)\,\vec{r}/f]_\Gamma := \begin{cases} \mathsf{union}_t\,(i, w[\vec{r}/f]_\Gamma, \vec{\mathbf{b}}) & \text{if } i = j \\ \mathsf{union}_t\,(i, (((\overline{w}\vec{\mathbf{b}})_{[0,\,s)})^{\tau_i}_\Gamma)[\vec{r}/f]_\Gamma, (\overline{w}\vec{\mathbf{b}})^{f}_{[s,\,z)}) & \text{if } i \neq j \end{cases}$$

$$(\overline{\mathsf{union}_t}\,\vec{\mathbf{b}})[(\xrightarrow{\mathsf{union}\ t}_q i)\,\vec{r}/f]_\Gamma := \mathsf{union}_t\,(i, ((\vec{\mathbf{b}}_{[0,\,s)})^{\tau_i}_\Gamma)[\vec{r}/f]_\Gamma, \vec{\mathbf{b}}^{f}_{[s,\,z)})$$

In the last two cases we have $\Gamma\,t = \vec{\tau}$, $s := \mathsf{bitsizeof}_\Gamma\,\tau_i$ and $z := \mathsf{bitsizeof}_\Gamma\,(\mathsf{union}\ t)$. The result of $w[\vec{r}/f]_\Gamma$ is only well-defined in case $w[\vec{r}]_\Gamma \neq \bot$.

Definition 6.35 Given a function $f : \mathsf{mtree} \to \mathsf{mtree}$, the *alter operation on memories* $(\_)[\_/f]_\Gamma : \mathsf{mem} \to \mathsf{addr} \to \mathsf{mem}$ is defined as:

$$m[a/f]_\Gamma := \begin{cases} m[(\mathsf{index}\ a) := (w[\mathsf{ref}_\Gamma\ a/\overline{f}]_\Gamma,\ \mu)] & \text{if } a \text{ is a byte address} \\ m[(\mathsf{index}\ a) := (w[\mathsf{ref}_\Gamma\ a/f]_\Gamma,\ \mu)] & \text{if } a \text{ is not a byte address} \end{cases}$$

provided that $m\,(\mathsf{index}\ a) = (w,\ \mu)$. In this definition we let:

$$\overline{f}\ w := (\overline{w}_{[0,\,i)}\ \overline{f\ (\overline{w}_{[i,\,j)})_\Gamma^{\mathsf{unsigned\ char}}}\ \overline{w}_{[j,\,\mathsf{bitsizeof}_\Gamma\ (\mathsf{typeof}\ w))})_\Gamma^{\mathsf{typeof}\ w}$$

where $i := \mathsf{byte}_\Gamma\ a \cdot \mathsf{char\_bits}$ and $j := (\mathsf{byte}_\Gamma\ a + 1) \cdot \mathsf{char\_bits}$.

The lookup and alter operation enjoy various properties; they preserve typing and satisfy laws about their interaction. We list some for illustration.

Lemma 6.36 (Alter commutes) *If* $\Gamma, \Delta \vdash m$, $a_1 \perp_\Gamma a_2$ *with:*

- $\Gamma, \Delta \vdash a_1 : \tau_1$, $m[a_1]_\Gamma = w_1$, *and* $\Gamma, \Delta \vdash f_1\ w_1 : \tau_1$, *and*
- $\Gamma, \Delta \vdash a_2 : \tau_2$, $m[a_2]_\Gamma = w_2$, *and* $\Gamma, \Delta \vdash f_2\ w_2 : \tau_2$,

then we have:

$$m[a_2/f_2]_\Gamma[a_1/f_1]_\Gamma = m[a_1/f_1]_\Gamma[a_2/f_2]_\Gamma.$$

Lemma 6.37 *If* $\Gamma, \Delta \vdash m$, $m[a]_\Gamma = w$, *and* a *is not a byte address, then:*

$$(m[a/f]_\Gamma)[a]_\Gamma = f\ w.$$

A variant of Lemma 6.37 for byte addresses is more subtle because a byte address can be used to modify padding. Since modifications of padding are masked, a successive lookup may yield a memory tree with more indeterminate bits. In Section 7.2 we present an alternative lemma that covers this situation.

We conclude this section with a useful helper function that *zips* a memory tree and a list. It is used in for example Definitions 6.58 and 7.22.

Definition 6.38 Given a function $f : \mathsf{pbit} \to B \to \mathsf{pbit}$, the operation that zips the leaves $\hat{f} : \mathsf{mtree} \to \mathsf{list}\ B \to \mathsf{mtree}$ is defined as:

$$\begin{aligned}
\hat{f}\ (\mathsf{base}_{\tau_\mathsf{b}}\ \vec{\mathbf{b}})\ \vec{y} &:= \mathsf{base}_{\tau_\mathsf{b}}\ (f\ \vec{\mathbf{b}}\ \vec{y}) \\
\hat{f}\ (\mathsf{array}_\tau\ \vec{w})\ \vec{y} &:= \mathsf{array}_\tau\ (\hat{f}\ w_0\ \vec{y}_{[0,\,s_1)} \ldots \hat{f}\ w_{n-1}\ \vec{y}_{[s_{n-1},\,s_n)}) \\
&\quad \text{where } n := |\vec{w}| \text{ and } s_i := \Sigma_{j<i} |\overline{w_j}| \\
\hat{f}\ (\mathsf{struct}_t\ \overrightarrow{w\vec{\mathbf{b}}})\ \vec{y} &:= \mathsf{struct}_t \begin{pmatrix} \hat{f}\ w_0\ \vec{y}_{[0,\,s_0)}\ f\ \vec{\mathbf{b}}_0\ \vec{y}_{[s_0,\,z_1)} \\ \ldots \\ \hat{f}\ w_{n-1}\ \vec{y}_{[z_{n-1},\,z_{n-1}+s_{n-1})}\ f\ \vec{\mathbf{b}}_{n-1}\ \vec{y}_{[z_{n-1}+s_{n-1},\,z_n)} \end{pmatrix} \\
&\quad \text{where } n := |\vec{w}|,\ s_i := |\overline{w_i}|, \text{ and } z_i := \Sigma_{j<i} |\overline{w_i}\ \vec{\mathbf{b}}_i| \\
\hat{f}\ (\mathsf{union}_t\ (i, w, \vec{\mathbf{b}}))\ \vec{y} &:= \mathsf{union}_t\ (i, \hat{f}\ w\ \vec{y}_{[0,\,|\overline{w}|)}, f\ \vec{\mathbf{b}}\ \vec{y}_{[|\overline{w}|,\,|\overline{w}\,\vec{\mathbf{b}}|)}) \\
\hat{f}\ (\overline{\mathsf{union}_t}\ \vec{\mathbf{b}})\ \vec{y} &:= \overline{\mathsf{union}_t}\ (f\ \vec{\mathbf{b}}\ \vec{y})
\end{aligned}$$

6.4 Representation of values

Memory trees (Definition 6.23) are still rather low-level and expose permissions and implementation specific properties such as bit representations. In this section we define *abstract values*, which are like memory trees but have mathematical integers and pointers instead of bit representations as leaves. Abstract values are used in the external interface of the memory model.

Definition 6.39 *Base values* are inductively defined as:

$$v_\mathsf{b} \in \mathsf{baseval} ::= \mathsf{indet}\,\tau_\mathsf{b} \mid \mathsf{nothing} \mid \mathsf{int}_{\tau_\mathsf{i}}\,x \mid \mathsf{ptr}\,p \mid \mathsf{byte}\,\vec{b}.$$

While performing byte-wise operations (for example, byte-wise copying a struct containing pointer values), abstraction is broken, and pointer fragment bits have to reside outside of memory. The value $\mathsf{byte}\,\vec{b}$ is used for this purpose.

Definition 6.40 The judgment $\Gamma, \Delta \vdash_\mathsf{b} v_\mathsf{b} : \tau_\mathsf{b}$ describes that *the base value* v_b *has base type* τ_b. It is inductively defined as:

$$\frac{\Gamma \vdash_\mathsf{b} \tau_\mathsf{b} \qquad \tau_\mathsf{b} \neq \mathsf{void}}{\Gamma, \Delta \vdash_\mathsf{b} \mathsf{indet}\,\tau_\mathsf{b} : \tau_\mathsf{b}} \qquad \frac{}{\Gamma, \Delta \vdash_\mathsf{b} \mathsf{nothing} : \mathsf{void}} \qquad \frac{x : \tau_\mathsf{i}}{\Gamma, \Delta \vdash_\mathsf{b} \mathsf{int}_{\tau_\mathsf{i}}\,x : \tau_\mathsf{i}}$$

$$\frac{\Gamma, \Delta \vdash_* p : \sigma_\mathsf{p}}{\Gamma, \Delta \vdash_\mathsf{b} \mathsf{ptr}\,p : \sigma_\mathsf{p}*} \qquad \frac{\Gamma, \Delta \vdash \vec{b} \quad |\vec{b}| = \mathsf{char\_bits} \quad \text{Not } \vec{b} \text{ all in } \{0,1\} \quad \text{Not } \vec{b} \text{ all } \lightning}{\Gamma, \Delta \vdash_\mathsf{b} \mathsf{byte}\,\vec{b} : \mathsf{unsigned\ char}}$$

The side-conditions of the typing rule for $\mathsf{byte}\,\vec{b}$ ensure canonicity of representations of base values. It ensures that the construct $\mathsf{byte}\,\vec{b}$ is only used if $\vec{b}$ cannot be represented as an integer $\mathsf{int}_{\mathsf{unsigned\ char}}\,x$ or $\mathsf{indet}\,(\mathsf{unsigned\ char})$.

In Definition 6.44 we define abstract values by extending base values with constructs for arrays, structs and unions. In order to define the operations to look up and store values in memory, we define conversion operations between abstract values and memory trees. Recall that the leaves of memory trees, which represent base values, are just sequences of bits. We therefore first define operations that convert base values to and from bits. These operations are called flatten and unflatten.

Definition 6.41 The *flatten operation* $\overline{(\_)}^\Gamma : \mathsf{baseval} \to \mathsf{list\ bit}$ is defined as:

$$\begin{aligned}
\overline{\mathsf{indet}\,\tau_\mathsf{b}}^\Gamma &:= \lightning^{\mathsf{bitsizeof}_\Gamma\,\tau_\mathsf{b}} \\
\overline{\mathsf{nothing}}^\Gamma &:= \lightning^{\mathsf{bitsizeof}_\Gamma\,\mathsf{void}} \\
\overline{\mathsf{int}_{\tau_\mathsf{i}}\,x}^\Gamma &:= \overline{x : \tau_\mathsf{i}} \\
\overline{\mathsf{ptr}\,p}^\Gamma &:= (\mathsf{ptr}\mid p \mid_\circ)_0 \ldots (\mathsf{ptr}\mid p \mid_\circ)_{\mathsf{bitsizeof}_\Gamma\,(\mathsf{typeof}\ p*)-1} \\
\overline{\mathsf{byte}\,\vec{b}}^\Gamma &:= \vec{b}
\end{aligned}$$

The operation $\overline{\_ : \tau_\mathsf{i}} : \mathbb{Z} \to \mathsf{list\ bool}$ is defined in Definition 4.4.

Definition 6.42 The *unflatten operation* $(\_)^\Gamma_{\tau_\mathsf{b}} : \mathsf{list\ bit} \to \mathsf{baseval}$ is defined as:

$$(\vec{b})^{\mathsf{void}}_\Gamma := \mathsf{nothing}$$

$$(\vec{b})^{\tau_\mathsf{i}}_\Gamma := \begin{cases} \mathsf{int}_{\tau_\mathsf{i}}\,(\vec{\beta})_{\tau_\mathsf{i}} & \text{if } \vec{b} \text{ is a } \{0,1\} \text{ sequence } \vec{\beta} \\ \mathsf{byte}\,\vec{b} & \text{if } \tau_\mathsf{i} = \mathsf{unsigned\ char}, \text{ not } \vec{b} \text{ all in } \{0,1\}, \text{ and not } \vec{b} \text{ all } \lightning \\ \mathsf{indet}\,\tau_\mathsf{i} & \text{otherwise} \end{cases}$$

$$(\vec{b})_\Gamma^{\sigma_\mathsf{p}*} := \begin{cases} \mathsf{ptr}\, p & \text{if } \vec{b} = (\mathsf{ptr}\, p)_0 \ldots (\mathsf{ptr}\, p)_{\mathsf{bitsizeof}_\Gamma\,(\sigma_\mathsf{p}*)-1} \text{ and } \mathsf{typeof}\, p = \sigma_\mathsf{p} \\ \mathsf{indet}\,(\sigma_\mathsf{p}*) & \text{otherwise} \end{cases}$$

The operation $(\_)_{\tau_\mathsf{i}} : \mathsf{list\ bool} \to \mathbb{Z}$ is defined in Definition 4.4.

The encoding of pointers is an important aspect of the flatten operation related to our treatment of effective types. Pointers are encoded as sequences of *frozen* pointer fragment bits $(\mathsf{ptr}\ |\, p\,|_\circ)_i$ (see Definition 6.5 for the definition of frozen pointers). Recall that the flatten operation is used to store base values in memory, whereas the unflatten operation is used to retrieve them. This means that whenever a pointer p is stored and read back, the frozen variant $|\, p\,|_\circ$ is obtained.

Lemma 6.43 *For each* $\Gamma, \Delta \vdash_\mathsf{b} v_\mathsf{b} : \tau_\mathsf{b}$ *we have* $(\overline{v_\mathsf{b}}^\Gamma)_\Gamma^{\tau_\mathsf{b}} = |\, v_\mathsf{b}\,|_\circ$.

Freezing formally describes the situations in which type-punning is allowed since a frozen pointer cannot be used to access a union of another variant than its current one (Definition 6.32). Let us consider an example:

```
union U { int x; short y; } u = { .x = 3 };
short *p = &u.y;  // a frozen version of the pointer &u.y is stored
printf("%d", *p); // type-punning via a frozen pointer -> undefined
```

Here, an attempt to type-punning is performed via the frozen pointer `p`, which is formally represented as:

$$(o_u : \mathsf{union\ U}, \xhookrightarrow{\mathsf{union\ U}}_\circ 1, 0)_{\mathsf{signed\ short} > * \mathsf{signed\ short}}.$$

The lookup operation on memory trees (which will be used to obtain the value of `*p` from memory, see Definitions 6.32 and 6.58) will fail. The annotation $\circ$ prevents a union from being accessed through an address to another variant than its current one. In the example below type-punning is allowed:

```
union U { int x; short y; } u = { .x = 3 };
printf("%d", u.y);
```

Here, type-punning is allowed because it is performed directly via `u.y`, which has not been stored in memory, and thus has not been frozen.

Definition 6.44 *Abstract values* are inductively defined as:

$$v \in \mathsf{val} ::= v_\mathsf{b} \mid \mathsf{array}_\tau\, \vec{v} \mid \mathsf{struct}_t\, \vec{v} \mid \mathsf{union}_t\,(i, v) \mid \overline{\mathsf{union}}_t\, \vec{v}.$$

The abstract value $\overline{\mathsf{union}}_t\, \vec{v}$ represents a union whose variant is unspecified. The values $\vec{v}$ correspond to interpretations of *all* variants of $\mathsf{union}\ t$. Consider:

```
union U { int x; short y; int *p; } u;
for (size_t i = 0; i < sizeof(u); i++) ((unsigned char*)&u)[i] = 0;
```

Here, the object representation of `u` is initialized with zeros, and its variant thus remains unspecified. The abstract value of `u` is[5]:

$$\overline{\mathsf{union}}_\mathsf{U}\,[\,\mathsf{int}_{\mathsf{signed\ int}}\,0,\ \mathsf{int}_{\mathsf{signed\ short}}\,0,\ \mathsf{indet}\,(\mathsf{signed\ int}*)\,]$$

[5] Note that the C11 standard does not guarantee that the `NULL` pointer is represented as zeros, thus `u.p` is not necessarily `NULL`.

Recall that the variants of a union occupy a single memory area, so the sequence $\vec{v}$ of a union value $\overline{\mathsf{union}}_t\ \vec{v}$ cannot be arbitrary. There should be a common bit sequence representing it. This is not the case in:

$$\overline{\mathsf{union}}_{\mathtt{U}}\,[\,\mathsf{int}_{\mathsf{signed\ int}}\,0,\ \mathsf{int}_{\mathsf{signed\ short}}\,1,\ \mathsf{indet}\,(\mathsf{signed\ int}*)\,]$$

The typing judgment for abstract values guarantees that $\vec{v}$ can be represented by a common bit sequence. In order to express this property, we first define the unflatten operation that converts a bit sequence into an abstract value.

Definition 6.45 The *unflatten operation* $(\_)^{\tau}_{\Gamma} : \mathsf{list\ bit} \to \mathsf{val}$ is defined as:

$$\begin{aligned}
(\vec{b})^{\tau_\mathsf{b}}_{\Gamma} &:= (\vec{b})^{\tau_\mathsf{b}}_{\Gamma} && \text{(the right hand side is Definition 6.41 on base values)}\\
(\vec{b})^{\tau[n]}_{\Gamma} &:= \mathsf{array}_{\tau}\,((\vec{b}_{[0,\,s)})^{\tau}_{\Gamma} \ldots (\vec{b}_{[(n-1)s,\,ns)})^{\tau}_{\Gamma}) && \text{where } s := \mathsf{bitsizeof}_{\Gamma}\,\tau\\
(\vec{b})^{\mathsf{struct}\ t}_{\Gamma} &:= \mathsf{struct}_{t}\,((\vec{b}_{[0,\,s_0)})^{\tau_0}_{\Gamma} \ldots (\vec{b}_{[z_{n-1},\,z_{n-1}+s_{n-1})})^{\tau_{n-1}}_{\Gamma}) &&\\
&\quad \text{where } \Gamma\,t = \vec{\tau},\ n := |\vec{\tau}|,\ s_i := \mathsf{bitsizeof}_{\Gamma}\,\tau_i \text{ and } z_i := \mathsf{bitoffsetof}_{\Gamma}\,\vec{\tau}\ i &&\\
(\vec{b})^{\mathsf{union}\ t}_{\Gamma} &:= \overline{\mathsf{union}}_{t}\,((\vec{b}_{[0,\,s_0)})^{\tau_0}_{\Gamma} \ldots (\vec{b}_{[0,\,s_{n-1})})^{\tau_{n-1}}_{\Gamma}) &&\\
&\quad \text{where } \Gamma\,t = \vec{\tau},\ n := |\vec{\tau}| \text{ and } s_i := \mathsf{bitsizeof}_{\Gamma}\,\tau_i &&
\end{aligned}$$

Definition 6.46 The judgment $\Gamma, \Delta \vdash v : \tau$ describes that *the value v has type τ*. It is inductively defined as:

$$\frac{\Gamma, \Delta \vdash_{\mathsf{b}} v_{\mathsf{b}} : \tau_{\mathsf{b}}}{\Gamma, \Delta \vdash v_{\mathsf{b}} : \tau_{\mathsf{b}}} \qquad \frac{\Gamma, \Delta \vdash \vec{v} : \tau \qquad |\vec{v}| = n \neq 0}{\Gamma, \Delta \vdash \mathsf{array}_{\tau}\,\vec{v} : \tau[n]}$$

$$\frac{\Gamma\,t = \vec{\tau} \qquad \Gamma, \Delta \vdash \vec{v} : \vec{\tau}}{\Gamma, \Delta \vdash \mathsf{struct}_t\,\vec{v} : \mathsf{struct}\ t} \qquad \frac{\Gamma\,t = \vec{\tau} \qquad i < |\vec{\tau}| \qquad \Gamma, \Delta \vdash v : \tau_i}{\Gamma, \Delta \vdash \mathsf{union}_t\,(i, v) : \mathsf{union}\ t}$$

$$\frac{\Gamma\,t = \vec{\tau} \qquad \Gamma, \Delta \vdash \vec{v} : \vec{\tau} \qquad \Gamma, \Delta \vdash \vec{b} \qquad \forall i\,.\,(v_i = (\vec{b}_{[0,\,\mathsf{bitsizeof}_{\Gamma}\,\tau_i)})^{\tau_i}_{\Gamma})}{\Gamma, \Delta \vdash \overline{\mathsf{union}}_t\,\vec{v} : \mathsf{union}\ t}$$

The flatten operation $\overline{(\_)}^{\Gamma} : \mathsf{val} \to \mathsf{list\ bit}$, which converts an abstract value v into a bit representation $\overline{v}^{\Gamma}$, is more difficult to define (we need this operation to define the conversion operation from abstract values into memory trees, see Definition 6.49). Since padding bits are not present in abstract values, we have to insert these. Also, in order to obtain the bit representation of an unspecified $\overline{\mathsf{union}}_t\,\vec{v}$ value, we have to *construct* the common bit sequence $\vec{b}$ representing $\vec{v}$. The typing judgment guarantees that such a sequence exists, but since it is not explicit in the value $\overline{\mathsf{union}}_t\,\vec{v}$, we have to reconstruct it from $\vec{v}$. Consider:

```
union U { struct S { short y; void *p; } x1; int x2; };
```

Assuming $\mathsf{sizeof}_{\Gamma}\,(\mathsf{signed\ int}) = \mathsf{sizeof}_{\Gamma}\,(\mathsf{any}*) = 4$ and $\mathsf{sizeof}_{\Gamma}\,(\mathsf{signed\ short}) = 2$, a well-typed `union U` value of an unspecified variant may be:

$$v = \overline{\mathsf{union}}_{\mathtt{U}}\,[\,\mathsf{struct}_{\mathtt{S}}\,[\,\mathsf{int}_{\mathsf{signed\ short}}\,0, \mathsf{ptr}\,p\,], \mathsf{int}_{\mathsf{signed\ int}}\,0\,].$$

The flattened versions of the variants of v are:

$$\begin{aligned}
\overline{\mathsf{struct}_{\mathtt{S}}\,[\,\mathsf{int}_{\mathsf{signed\ short}}\,0, \mathsf{ptr}\,p\,]}^{\Gamma} &= 0\ldots0\ 0\ldots0\ \lightning\ldots\lightning\ \lightning\ldots\lightning\ (\mathsf{ptr}\ p)_0 \ldots (\mathsf{ptr}\ p)_{31}\\
\overline{\mathsf{int}_{\mathsf{signed\ int}}\,0}^{\Gamma} &= 0\ldots0\ 0\ldots0\ 0\ldots0\ 0\ldots0\\
\hline
\overline{v}^{\Gamma} &= 0\ldots0\ 0\ldots0\ 0\ldots0\ 0\ldots0\ (\mathsf{ptr}\ p)_0 \ldots (\mathsf{ptr}\ p)_{31}
\end{aligned}$$

This example already illustrates that so as to obtain the common bit sequence $\overline{v}^\Gamma$ of v we have to insert padding bits and "join" the padded bit representations.

Definition 6.47 The *join operation on bits* $\sqcup : \mathsf{bit} \to \mathsf{bit} \to \mathsf{bit}$ is defined as:

$$\lightning \sqcup b := b \qquad b \sqcup \lightning := b \qquad b \sqcup b := b.$$

Definition 6.48 The *flatten operation* $\overline{(\_)}^\Gamma : \mathsf{val} \to \mathsf{list\ bit}$ is defined as:

$$\begin{aligned}
\overline{v_\mathsf{b}}^\Gamma &:= \overline{v_\mathsf{b}}^\Gamma \\
\overline{\mathsf{array}_\tau\ \vec{v}}^\Gamma &:= \overline{v_0}^\Gamma \ldots \overline{v_{|\vec{v}|-1}}^\Gamma \\
\overline{\mathsf{struct}_t\ \vec{v}}^\Gamma &:= (\overline{v_0}^\Gamma \lightning^\infty)_{[0,\,z_0)} \ldots (\overline{v_{n-1}}^\Gamma \lightning^\infty)_{[0,\,z_{n-1})} \\
&\quad \text{where } \Gamma\, t = \vec{\tau},\ n := |\vec{\tau}|, \text{ and } z_i := \mathsf{bitoffsetof}_\Gamma\ \vec{\tau}\ i \\
\overline{\mathsf{union}_t\ (i, v)}^\Gamma &:= (\overline{v}^\Gamma \lightning^\infty)_{[0,\,\mathsf{bitsizeof}_\Gamma\ (\mathsf{union}\ t))} \\
\overline{\overline{\mathsf{union}_t}\ \vec{v}}^\Gamma &:= \bigsqcup_{i=0}^{|\vec{v}|-1} (\overline{v_i}^\Gamma \lightning^\infty)_{[0,\,\mathsf{bitsizeof}_\Gamma\ (\mathsf{union}\ t))}
\end{aligned}$$

The operation $\mathsf{ofval}_\Gamma : \mathsf{list\ perm} \to \mathsf{val} \to \mathsf{mtree}$, which converts a value v of type τ into a memory tree $\mathsf{ofval}_\Gamma\ \vec{\gamma}\ v$, is albeit technical fairly straightforward. In principle it is just a recursive definition that uses the flatten operation $\overline{v_\mathsf{b}}^\Gamma$ for base values v_b and the flatten operation $\overline{\overline{\mathsf{union}_t}\ \vec{v}}^\Gamma$ for unions $\overline{\mathsf{union}_t}\ \vec{v}$ of an unspecified variant.

The technicality is that abstract values do not contain permissions, so we have to merge the given value with permissions. The sequence $\vec{\gamma}$ with $|\vec{\gamma}| = \mathsf{bitsizeof}_\Gamma\ \tau$ represents a flattened sequence of permissions. In the definition of the memory store $m\langle a := v\rangle_\Gamma$ (see Definition 6.58), we convert v into the stored memory tree $\mathsf{ofval}_\Gamma\ \vec{\gamma}\ v$ where γ constitutes the old permissions of the object at address a.

Definition 6.49 The operation $\mathsf{ofval}_\Gamma : \mathsf{list\ perm} \to \mathsf{val} \to \mathsf{mtree}$ is defined as:

$$\begin{aligned}
\mathsf{ofval}_\Gamma\ \vec{\gamma}\ (v_\mathsf{b}) &:= \mathsf{base}_{\mathsf{typeof}\ v_\mathsf{b}}\ \overrightarrow{\gamma b} \quad \text{where } \vec{b} := \overline{v_\mathsf{b}}^\Gamma \\
\mathsf{ofval}_\Gamma\ \vec{\gamma}\ (\mathsf{array}_\tau\ \vec{v}) &:= \mathsf{array}_\tau\ (\mathsf{ofval}_\Gamma\ \vec{\gamma}_{[0,\,s)}\ v_0 \ldots \mathsf{ofval}_\Gamma\ \vec{\gamma}_{[(n-1)s,\,ns)}\ v_{n-1}) \\
&\quad \text{where } s := \mathsf{bitsizeof}_\Gamma\ \tau \text{ and } n := |\vec{v}| \\
\mathsf{ofval}_\Gamma\ \vec{\gamma}\ (\mathsf{struct}_t\ \vec{v}) &:= \mathsf{struct}_t \begin{pmatrix} \mathsf{ofval}_\Gamma\ \vec{\gamma}_{[0,\,s_0)}\ v_0\ \vec{\gamma}^{\lightning}_{[s_0,\,z_1)} \\ \ldots \\ \mathsf{ofval}_\Gamma\ \vec{\gamma}_{[z_{n-1},\,z_{n-1}+s_{n-1})}\ v_{n-1}\ \vec{\gamma}^{\lightning}_{[z_{n-1}+s_{n-1},\,z_n)} \end{pmatrix} \\
&\quad \text{where } \Gamma\, t = \vec{\tau},\ n := |\vec{\tau}|,\ s_i := \mathsf{bitsizeof}_\Gamma\ \tau_i \\
&\quad \text{and } z_i := \mathsf{bitoffsetof}_\Gamma\ \vec{\tau}\ i \\
\mathsf{ofval}_\Gamma\ \vec{\gamma}\ (\mathsf{union}_t\ (i, v)) &:= \mathsf{union}_t\ (i, \mathsf{ofval}_\Gamma\ \vec{\gamma}_{[0,\,s)}\ v, \vec{\gamma}^{\lightning}_{[s,\,\mathsf{bitsizeof}_\Gamma\ (\mathsf{union}\ t))}) \\
&\quad \text{where } s := \mathsf{bitsizeof}_\Gamma\ (\mathsf{typeof}\ v) \\
\mathsf{ofval}_\Gamma\ \vec{\gamma}\ (\overline{\mathsf{union}_t}\ \vec{v}) &:= \overline{\mathsf{union}_t}\ \overrightarrow{\gamma b} \quad \text{where } \vec{b} := \overline{\overline{\mathsf{union}_t}\ \vec{v}}^\Gamma
\end{aligned}$$

Converting a memory tree into a value is as expected: permissions are removed and unions are interpreted as values corresponding to each variant.

Definition 6.50 The operation $\mathsf{toval}_\Gamma : \mathsf{mtree} \to \mathsf{val}$ is defined as:

$$\begin{aligned}
\mathsf{toval}_\Gamma\,(\mathsf{base}_{\tau_\mathsf{b}}\,\overrightarrow{\gamma b}) &:= (\vec{b})_\Gamma^{\tau_\mathsf{b}}\\
\mathsf{toval}_\Gamma\,(\mathsf{array}_\tau\,\vec{w}) &:= \mathsf{array}_\tau\,(\mathsf{toval}_\Gamma\,w_0 \ldots \mathsf{toval}_\Gamma\,w_{|\vec{w}|-1})\\
\mathsf{toval}_\Gamma\,(\mathsf{struct}_t\,\overrightarrow{w\mathbf{b}}) &:= \mathsf{struct}_t\,(\mathsf{toval}_\Gamma\,w_0 \ldots \mathsf{toval}_\Gamma\,w_{|\vec{w}|-1})\\
\mathsf{toval}_\Gamma\,(\mathsf{union}_t\,(i, w, \vec{\mathbf{b}})) &:= \mathsf{union}_t\,(i, \mathsf{toval}_\Gamma\,w)\\
\mathsf{toval}_\Gamma\,(\overline{\mathsf{union}_t}\,\overrightarrow{\gamma b}) &:= (\vec{b})_\Gamma^{\mathsf{union}\ t}
\end{aligned}$$

The function toval_Γ is an inverse of ofval_Γ up to freezing of pointers. Freezing is intended, it makes indirect type-punning illegal.

Lemma 6.51 *Given $\Gamma, \Delta \vdash v : \tau$, and let $\vec{\gamma}$ be a flattened sequence of permissions with $|\vec{\gamma}| = \mathsf{bitsizeof}_\Gamma\,\tau$, then we have:*

$$\mathsf{toval}_\Gamma\,(\mathsf{ofval}_\Gamma\,\vec{\gamma}\,v) = |\,v\,|_\circ.$$

The other direction does not hold because invalid bit representations will become indeterminate values.

```
struct S { int *p; } s;
for (size_t i = 0; i < sizeof(s); i++) ((unsigned char*)&s)[i] = i;
// s has some bit representation that does not constitute a pointer
struct S s2 = s;
// After reading s, and storing it, there are no guarantees about s2,
// whose object representation thus consists of ↯s
```

We finish this section by defining the indeterminate abstract value $\mathsf{new}_\Gamma\,\tau$, which consists of indeterminate base values. The definition is similar to its counterpart on memory trees (Definition 6.31).

Definition 6.52 The operation $\mathsf{new}_\Gamma : \mathsf{type} \to \mathsf{val}$ that yields the indeterminate value is defined as:

$$\mathsf{new}_\Gamma\,\tau := (\lightning^{\mathsf{bitsizeof}_\Gamma\,\tau})_\Gamma^\tau.$$

Lemma 6.53 *If $\Gamma \vdash \tau$, then:*

$$\mathsf{toval}_\Gamma\,(\mathsf{new}_\Gamma^\gamma\,\tau) = \mathsf{new}_\Gamma\,\tau \quad \text{and} \quad \mathsf{ofval}_\Gamma\,(\gamma^{\mathsf{bitsizeof}_\Gamma\,\tau})\,(\mathsf{new}_\Gamma\,\tau) = \mathsf{new}_\Gamma^\gamma\,\tau.$$

6.5 Memory operations

Now that we have all primitive definitions in place, we can compose these to implement the actual memory operations as described in the beginning of this section. The last part that is missing is a data structure to keep track of objects that have been locked. Intuitively, this data structure should represent a set of addresses, but up to overlapping addresses.

Definition 6.54 *Locksets* are defined as:

$$\Omega \in \mathsf{lockset} := \mathcal{P}_\mathsf{fin}(\mathsf{index} \times \mathbb{N}).$$

Elements of locksets are pairs (o, i) where $o \in \mathsf{index}$ describes the object identifier and $i \in \mathbb{N}$ a bit-offset in the object described by o. We introduce a typing judgment to describe that the structure of locksets matches up with the memory layout.

Definition 6.55 The judgment $\Gamma, \Delta \vdash \Omega$ describes that *the lockset* Ω *is valid*. It is inductively defined as:

$$\frac{\text{for each} \quad (o,\, i) \in \Omega \quad \text{there is a } \tau \text{ with} \quad \Delta \vdash o : \tau \text{ and } i < \mathsf{bitsizeof}_\Gamma\, \tau}{\Gamma, \Delta \vdash \Omega}$$

Definition 6.56 The *singleton lockset* $\{\_\}_\Gamma : \mathsf{addr} \to \mathsf{lockset}$ is defined as:

$$\{a\}_\Gamma := \{(\mathsf{index}\ a, i) \mid \mathsf{bitoffset}_\Gamma\, a \le i < \mathsf{bitoffset}_\Gamma\, a + \mathsf{bitsizeof}_\Gamma\, (\mathsf{typeof}\ a)\}.$$

Lemma 6.57 *If* $\Gamma, \Delta \vdash a_1 : \sigma_1$ *and* $\Gamma, \Delta \vdash a_2 : \sigma_2$ *and* $\Gamma \vdash \{a_1, a_2\}\ \mathsf{strict}$*, then:*

$$a_1 \perp_\Gamma a_2 \quad \text{implies} \quad \{a_1\}_\Gamma \cap \{a_2\}_\Gamma = \emptyset.$$

Definition 6.58 The *memory operations* are defined as:

$$\begin{aligned}
m\langle a\rangle_\Gamma &:= \mathsf{toval}_\Gamma\, w \quad \text{if } m[a]_\Gamma = w \text{ and } \forall i\,.\, \mathsf{Readable} \subseteq \mathsf{kind}\ (\overline{w})_i \\
\mathsf{force}_\Gamma\, a\, m &:= m[(\mathsf{index}\ a) := (w[\mathsf{ref}_\Gamma\, a/\lambda w'\,.\, w']_\Gamma,\, \mu)] \quad \text{if } m\,(\mathsf{index}\ a) = (w,\, \mu) \\
m\langle a := v\rangle_\Gamma &:= m[a/\lambda w\,.\, \mathsf{ofval}_\Gamma\, (\overline{w}_{\mathbf{1}})\, v]_\Gamma \\
\mathsf{writable}_\Gamma\, a\, m &:= \exists w\,.\, m[a]_\Gamma = w \text{ and } \forall i\,.\, \mathsf{Writable} \subseteq \mathsf{kind}\ (\overline{w})_i \\
\mathsf{lock}_\Gamma\, a\, m &:= m[a/\lambda w\,.\ \text{apply } \mathsf{lock} \text{ to all permissions of } w]_\Gamma \\
\mathsf{unlock}\ \Omega\, m &:= \{(o,\, (\hat{f}\, w\, \vec{y},\, \mu)) \mid m\, o = (w,\, \mu)\} \cup \{(o,\, \tau) \mid m\, o = \tau\} \\
&\quad \text{where } f\,(\gamma,\, b)\,\mathsf{true} := (\mathsf{unlock}\ \gamma,\, b) \\
&\qquad\quad f\,(\gamma,\, b)\,\mathsf{false} := (\gamma,\, b), \\
&\quad \text{and } \vec{y} := ((o,\, 0) \in \Omega) \ldots ((o,\, |\mathsf{bitsizeof}_\Gamma\, (\mathsf{typeof}\ w)| - 1) \in \Omega) \\
\mathsf{alloc}_\Gamma\, o\, v\, \mu\, m &:= m[o := (\mathsf{ofval}_\Gamma\, (\Diamond(0,\, 1)^{\mathsf{bitsizeof}_\Gamma\, (\mathsf{typeof}\ v)})\, v,\, \mu)] \\
\mathsf{freeable}\ a\, m &:= \exists o\, \tau\, \sigma\, n\, w\,.\, a = (o : \tau, \xrightarrow{\tau[n]} 0, 0)_{\tau >_* \sigma},\ m\, o = (w,\, \mathsf{true}) \\
&\qquad\quad \text{and all } \overline{w} \text{ have the permission } \Diamond(0,\, 1) \\
\mathsf{free}\ o\, m &:= m[o := \mathsf{typeof}\ w] \quad \text{if } m\, o = (w,\, \mu)
\end{aligned}$$

The lookup operation $m\langle a\rangle_\Gamma$ uses the lookup operation $m[a]_\Gamma$ that yields a memory tree w (Definition 6.33), and then converts w into the value $\mathsf{toval}_\Gamma\, w$. The operation $m[a]_\Gamma$ already yields $\perp$ in case effective types are violated or a is an end-of-array address. The additional condition of $m\langle a\rangle_\Gamma$ ensures that the permissions allow for a read access. Performing a lookup affects the effective types of the object at address a. This is factored out by the operation $\mathsf{force}_\Gamma\, a\, m$ which applies the identity function to the subobject at address a in the memory m. Importantly, this does not change the memory contents, but merely changes the variants of the involved unions.

The store operation $m\langle a := v\rangle_\Gamma$ uses the alter operation $m[a/\lambda w\,.\, \mathsf{ofval}_\Gamma\, (\overline{w}_{\mathbf{1}})\, v]_\Gamma$ on memories (Definition 6.35) to apply $\lambda w\,.\, \mathsf{ofval}_\Gamma\, (\overline{w}_{\mathbf{1}})\, v$ to the subobject at address a. The stored value v is converted into a memory tree while retaining the permissions $\overline{w}_{\mathbf{1}}$ of the previously stored memory tree w at address a.

The definition of $\mathsf{lock}_\Gamma\, a\, m$ is straightforward. In the Coq development we use a map operation on memory trees to apply the function lock (Definition 5.5) to the permission of each bit of the memory tree at address a.

The operation $\mathsf{unlock}\ \Omega\ m$ unlocks a whole lockset Ω, rather than an individual address, in memory m. For each memory tree w at object identifier o, it converts Ω to a Boolean vector $\vec{y} = ((o, 0) \in \Omega) \ldots ((o, |\mathsf{bitsizeof}_\Gamma\ (\mathsf{typeof}\ w)| - 1) \in \Omega)$ and merges w with $\vec{y}$ (using Definition 6.38) to apply unlock (Definition 5.5) to the permissions of bits that should be unlocked in w. We show some lemmas to illustrate that the operations for locking and unlocking enjoy the intended behavior:

Lemma 6.59 *If* $\Gamma, \Delta \vdash m$ *and* $\Gamma, \Delta \vdash a : \tau$ *and* $\mathsf{writable}_\Gamma\ a\ m$*, then we have:*

$$\mathsf{locks}\ (\mathsf{lock}_\Gamma\ a\ m) = \mathsf{locks}\ m \cup \{a\}_\Gamma.$$

Lemma 6.60 *If* $\Omega \subseteq \mathsf{locks}\ m$*, then* $\mathsf{locks}\ (\mathsf{unlock}\ \Omega\ m) = \mathsf{locks}\ m \setminus \Omega$*.*

Provided $o \notin \mathsf{dom}\ m$, allocation $\mathsf{alloc}_\Gamma\ o\ v\ \mu\ m$ extends the memory with a new object holding the value v and *full* permissions $\Diamond(0, 1)$. Typically we use $v = \mathsf{new}_\Gamma\ \tau$ for some τ, but global and static variables are allocated with a specific value v.

The operation $\mathsf{free}\ o\ m$ deallocates the object o in m, and keeps track of the type of the deallocated object. In order to deallocate dynamically obtained memory via `free`, the side-condition $\mathsf{freeable}\ a\ m$ describes that the permissions are sufficient for deallocation, and that a points to the first element of a `malloc`ed array.

All operations preserve typing and satisfy the expected laws about their interaction. We list some for illustration.

Fact 6.61 If $\mathsf{writable}_\Gamma\ a\ m$, then there exists a value v with $a\langle m\rangle_\Gamma = v$.

Lemma 6.62 (Stores commute) *If* $\Gamma, \Delta \vdash m$ *and* $a_1 \perp_\Gamma a_2$ *with:*

- $\Gamma, \Delta \vdash a_1 : \tau_1$*,* $\mathsf{writable}_\Gamma\ a_1\ m$*, and* $\Gamma, \Delta \vdash v_1 : \tau_1$*, and*
- $\Gamma, \Delta \vdash a_2 : \tau_2$*,* $\mathsf{writable}_\Gamma\ a_2\ m$*, and* $\Gamma, \Delta \vdash v_2 : \tau_2$*,*

then we have:

$$m\langle a_2 := v_2\rangle_\Gamma\langle a_1 := v_1\rangle_\Gamma = m\langle a_1 := v_1\rangle_\Gamma\langle a_2 := v_2\rangle_\Gamma.$$

Lemma 6.63 (Lookup after store) *If* $\Gamma, \Delta \vdash m$ *and* $\Gamma, \Delta \vdash a : \tau$ *and* $\Gamma, \Delta \vdash v : \tau$ *and* $\mathsf{writable}_\Gamma\ a\ m$ *and* a *is not a byte address, then we have:*

$$(m\langle a := v\rangle_\Gamma)\langle a\rangle_\Gamma = |\, v \,|_\circ.$$

Storing a value v in memory and then retrieving it, does not necessarily yield the same value v. It intentionally yields the value $|\, v \,|_\circ$ whose pointers have been frozen. Note that the above result does not hold for byte addresses, which may store a value in a padding byte, in which case the resulting value is indeterminate.

Lemma 6.64 (Stores and lookups commute) *If* $\Gamma, \Delta \vdash m$ *and* $a_1 \perp_\Gamma a_2$ *and* $\Gamma, \Delta \vdash a_2 : \tau_2$ *and* $\mathsf{writable}_\Gamma\ a_2\ m$ *and* $\Gamma, \Delta \vdash v_2 : \tau_2$*, then we have:*

$$m\langle a_1\rangle_\Gamma = v_1 \quad \text{implies} \quad (m\langle a_2 := v_2\rangle_\Gamma)\langle a_1\rangle_\Gamma = v_1.$$

These results follow from Lemma 6.36, 6.37 and 6.51.

7 Formal proofs

7.1 Type-based alias analysis

The purpose of C11's notion of effective types [27, 6.5p6-7] is to make it possible for compilers to perform typed-based alias analysis. Consider:

```
short g(int *p, short *q) {
  short x = *q; *p = 10; return x;
}
```

Here, a compiler should be able to assume that `p` and `q` are not aliased because they point to objects with different types (although the integer types signed short and signed int may have the same representation, they have different integer ranks, see Definition 4.2, and are thus different types). If `g` is called with aliased pointers, execution of the function body should have undefined behavior in order to allow a compiler to soundly assume that `p` and `q` are not aliased.

From the C11 standard's description of effective types it is not immediate that calling `g` with aliased pointers results in undefined behavior. We prove an abstract property of our memory model that shows that this is indeed a consequence, and that indicates a compiler can perform type-based alias analysis. This also shows that our interpretation of effective types of the C11 standard, in line with the interpretation from the GCC documentation [20], is sensible.

Definition 7.1 A type τ is a *subobject type of* σ, notation $\tau \subseteq_\Gamma \sigma$, if there exists some reference $\vec{r}$ with $\Gamma \vdash \vec{r} : \sigma \rightarrowtail \tau$.

For example, `int[2]` is a subobject type of `struct S { int x[2]; int y[3]; }` and `int[2][2]`, but not of `struct S { short x[2]; }`, nor of `int(*)[2]`.

Theorem 7.2 (Strict-aliasing) *Given* $\Gamma, \Delta \vdash m$*, frozen addresses* a_1 *and* a_2 *with* $\Delta, m \vdash a_1 : \sigma_1$ *and* $\Delta, m \vdash a_2 : \sigma_2$ *and* $\sigma_1, \sigma_2 \neq$ unsigned char*, then either:*

1. *We have* $\sigma_1 \subseteq_\Gamma \sigma_2$ *or* $\sigma_2 \subseteq_\Gamma \sigma_1$*.*
2. *We have* $a_1 \perp_\Gamma a_2$*.*
3. *Accessing* a_1 *after accessing* a_2 *and vice versa fails. That means:*
   (a) $(\mathsf{force}_\Gamma\, a_2\, m)\langle a_1\rangle_\Gamma = \bot$ *and* $(\mathsf{force}_\Gamma\, a_1\, m)\langle a_2\rangle_\Gamma = \bot$*, and*
   (b) $m\langle a_2 := v_1\rangle_\Gamma\langle a_1\rangle_\Gamma = \bot$ *and* $m\langle a_1 := v_2\rangle_\Gamma\langle a_2\rangle_\Gamma = \bot$ *for all stored values* v_1 *and* v_2*.*

This theorem implies that accesses to addresses of disjoint type are either non-overlapping or have undefined behavior. Fact 6.61 accounts for a store after a lookup. Using this theorem, a compiler can optimize the generated code in the example based on the assumption that `p` and `q` are not aliased. Reconsider:

```
short g(int *p, short *q) { short x = *q; *p = 10; return x; }
```

If `p` and `q` are aliased, then calling `g` yields undefined behavior because the assignment `*p = 10` violates effective types. Let m be the initial memory while executing `g`, and let $a_{\mathtt{p}}$ and $a_{\mathtt{q}}$ be the addresses corresponding to `p` and `q`, then the condition $\mathsf{writable}_\Gamma\, a_{\mathtt{p}}\, (\mathsf{force}_\Gamma\, a_{\mathtt{q}}\, m)$ does not hold by Theorem 7.2 and Fact 6.61.

7.2 Memory refinements

This section defines the notion of *memory refinements* that allows us to relate memory states. The author's PhD thesis [33] shows that the CH_2O operational semantics is invariant under this notion. Memory refinements form a general way to validate many common-sense properties of the memory model in a formal way. For example, they show that the memory is invariant under relabeling. More interestingly, they show that symbolic information (such as variants of unions) cannot be observed.

Memory refinements also open the door to reason about program transformations. We demonstrate their usage by proving soundness of constant propagation and by verifying an abstract version of `memcpy`.

Memory refinements are a variant of Leroy and Blazy's notion of memory extensions and injections [41]. A memory refinement is a relation $m_1 \sqsubseteq^f_\Gamma m_2$ between a source memory state m_1 and target memory state m_2, where:

1. The function $f : \mathsf{index} \to \mathsf{option}\ (\mathsf{index} \times \mathsf{ref})$ is used to rename object identifiers and to coalesce multiple objects into subobjects of a compound object.
2. Deallocated objects in m_1 may be replaced by arbitrary objects in m_2.
3. Indeterminate bits $\lightning$ in m_1 may be replaced by arbitrary bits in m_2.
4. Pointer fragment bits $(\mathsf{ptr}\ p)_i$ that belong to deallocated pointers in m_1 may be replaced by arbitrary bits in m_2.
5. Effective types may be weakened. That means, unions with a specific variant in m_1 may be replaced by unions with an unspecified variant in m_2, and pointers with frozen union annotations $\circ$ in m_1 may be replaced by pointers with unfrozen union annotations $\bullet$ in m_2.

The key property of a memory refinement $m_1 \sqsubseteq^f_\Gamma m_2$, as well as of Leroy and Blazy's memory extensions and injections, is that memory operations are more defined on the target memory m_2 than on the source memory m_1. For example, if a lookup succeeds on m_1, it also succeed on m_2 and yield a related value.

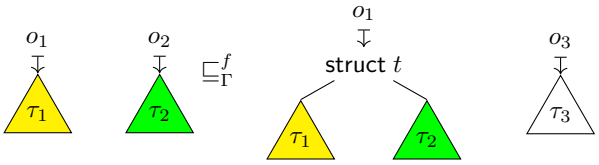
The main judgment $m_1 \sqsubseteq^{f:\Delta_1 \mapsto \Delta_2}_\Gamma m_2$ of memory refinements will be built using a series of refinement relations on the structures out of which the memory consists (addresses, pointers, bits, memory trees, values). All of these judgments should satisfy some basic properties, which are captured by the judgment $\Delta_1 \sqsubseteq^f_\Delta \Delta_2$.

Definition 7.3 A *renaming function* $f : \mathsf{index} \to \mathsf{option}\ (\mathsf{index} \times \mathsf{ref})$ is a *refinement*, notation $\Delta_1 \sqsubseteq^f_\Delta \Delta_2$, if the following conditions hold:

1. If $f\, o_1 = (o, \vec{r}_1)$ and $f\, o_2 = (o, \vec{r}_2)$, then $o_1 = o_2$ or $\vec{r}_1 \perp \vec{r}_2$ (*injectivity*).
2. If $f\, o_1 = (o_2, \vec{r})$, then $\mathsf{frozen}\ \vec{r}$.
3. If $f\, o_1 = (o_2, \vec{r})$ and $\Delta_1 \vdash o_1 : \sigma$, then $\Delta_2 \vdash o_2 : \tau$ and $\Gamma \vdash \vec{r} : \tau \rightarrowtail \sigma$ for a τ.
4. If $f\, o_1 = (o_2, \vec{r})$ and $\Delta_2 \vdash o_2 : \tau$, then $\Delta_1 \vdash o_1 : \sigma$ and $\Gamma \vdash \vec{r} : \tau \rightarrowtail \sigma$ for a σ.
5. If $f\, o_1 = (o_2, \vec{r})$ and $\Delta_1 \vdash o_1\ \mathsf{alive}$, then $\Delta_2 \vdash o_2\ \mathsf{alive}$.

The renaming function $f : \mathsf{index} \to \mathsf{option}\ (\mathsf{index} \times \mathsf{ref})$ is the core of all refinement judgments. It is used to rename object identifiers and to coalesce multiple source objects into subobjects of a single compound target object.

Consider a renaming f with $f\,o_1 = (o_1, \xhookrightarrow{\mathsf{struct}\ t} 0)$ and $f\,o_2 = (o_1, \xhookrightarrow{\mathsf{struct}\ t} 1)$, and an environment Γ with $\Gamma\,t = [\,\tau_1, \tau_2\,]$. This gives rise to following refinement:



Injectivity of renaming functions guarantees that distinct source objects are coalesced into disjoint target subobjects. In the case of Blazy and Leroy, the renaming functions have type $\mathsf{index} \to \mathsf{option}\ (\mathsf{index} \times \mathbb{N})$, but we replaced the natural number by a reference since our memory model is structured using trees.

Since memory refinements rearrange the memory layout, addresses should be rearranged accordingly. The judgment $a_1 \sqsubseteq_\Gamma^{f:\Delta_1 \mapsto \Delta_2} a_2 : \tau_\mathsf{p}$ describes how a_2 is obtained by renaming a_1 according to the renaming f, and moreover allows frozen union annotations $\circ$ in a_1 to be changed into unfrozen ones $\bullet$ in a_2. The index τ_p in the judgment $a_1 \sqsubseteq_\Gamma^{f:\Delta_1 \mapsto \Delta_2} a_2 : \tau_\mathsf{p}$ corresponds to the type of a_1 and a_2.

The judgment for addresses is lifted to the judgment for pointers in the obvious way. The judgment for bits is inductively defined as:

$$\frac{\beta \in \{0,1\}}{\beta \sqsubseteq_\Gamma^{f:\Delta_1 \mapsto \Delta_2} \beta} \qquad \frac{p_1 \sqsubseteq_\Gamma^{f:\Delta_1 \mapsto \Delta_2} p_2 : \sigma_\mathsf{p} \qquad \mathsf{frozen}\ p_2 \qquad i < \mathsf{bitsizeof}_\Gamma\,(\sigma_\mathsf{p}*)}{(\mathsf{ptr}\ p_1)_i \sqsubseteq_\Gamma^{f:\Delta_1 \mapsto \Delta_2} (\mathsf{ptr}\ p_2)_i}$$

$$\frac{\Gamma, \Delta_2 \vdash b}{\lightning \sqsubseteq_\Gamma^{f:\Delta_1 \mapsto \Delta_2} b} \qquad \frac{\Gamma, \Delta_1 \vdash a : \sigma \qquad \Delta_1 \not\vdash a\ \mathsf{alive} \qquad \Gamma, \Delta_2 \vdash b}{(\mathsf{ptr}\ a)_i \sqsubseteq_\Gamma^{f:\Delta_1 \mapsto \Delta_2} b}$$

The last two rules allow indeterminate bits $\lightning$, as well as pointer fragment bits $(\mathsf{ptr}\ a)_i$ belonging to deallocated storage, to be replaced by arbitrary bits b.

The judgment is lifted to memory trees following the tree structure and using the following additional rule:

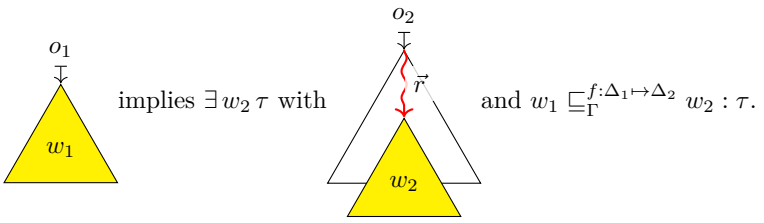
$$\frac{\begin{array}{c}\Gamma\,t = \vec{\tau} \qquad \Gamma, \Delta \vdash w_1 : \tau_i \qquad \overline{w_1}\,\vec{\mathbf{b}}_1 \sqsubseteq_\Gamma^{f:\Delta_1 \mapsto \Delta_2} \vec{\mathbf{b}}_2 \qquad \vec{\mathbf{b}}_1\ \mathrm{all}\ \lightning \\ \mathsf{bitsizeof}_\Gamma\,(\mathsf{union}\ t) = \mathsf{bitsizeof}_\Gamma\,\tau_i + |\vec{\mathbf{b}}_1| \qquad \neg\mathsf{unmapped}\ (\overline{w}_1\,\vec{\mathbf{b}}_1)\end{array}}{\mathsf{union}_t\,(i, w_1, \vec{\mathbf{b}}_1) \sqsubseteq_\Gamma^{f:\Delta_1 \mapsto \Delta_2} \overline{\mathsf{union}_t}\,\vec{\mathbf{b}}_2 : \mathsf{union}\ t}$$

This rule allows a union that has a specific variant in the source to be replaced by a union with an unspecified variant in the target. The direction seems counter intuitive, but keep in mind that unions with an unspecified variant allow more behaviors.

Lemma 7.4 *If* $w_1 \sqsubseteq_\Gamma^{f:\Delta_1 \mapsto \Delta_2} w_2 : \tau$*, then* $\Gamma, \Delta_1 \vdash w_1 : \tau$ *and* $\Gamma, \Delta_2 \vdash w_2 : \tau$*.*

This lemma is useful because it removes the need for simultaneous inductions on both typing and refinement judgments.

We define $m_1 \sqsubseteq^{f}_{\Gamma} m_2$ as $m_1 \sqsubseteq^{f:\overline{m_1}\mapsto\overline{m_2}}_{\Gamma} m_2$, where the judgment $m_1 \sqsubseteq^{f:\Delta_1\mapsto\Delta_2}_{\Gamma} m_2$ is defined such that if $f\, o_1 = (o_2, \vec{r})$, then:



The above definition makes sure that objects are renamed, and possibly coalesced into subobjects of a compound object, as described by the renaming function f.

In order to reason about program transformations modularly, we show that memory refinements can be composed.

Lemma 7.5 *Memory refinements are reflexive for valid memories, that means, if* $\Gamma, \Delta \vdash m$, *then* $m \sqsubseteq^{\mathsf{id}:\Delta\mapsto\Delta}_{\Gamma} m$ *where* $\mathsf{id}\, o := (o, \varepsilon)$.

Lemma 7.6 *Memory refinements compose, that means, if* $m_1 \sqsubseteq^{f:\Delta_1\mapsto\Delta_2}_{\Gamma} m_2$ *and* $m_2 \sqsubseteq^{f':\Delta_2\mapsto\Delta_3}_{\Gamma} m_3$, *then* $m_1 \sqsubseteq^{f'\circ f:\Delta_1\mapsto\Delta_3}_{\Gamma} m_3$ *where:*

$$(f' \circ f)\, o_1 := \begin{cases} (o_3, \vec{r}_2\, \vec{r}_3) & \text{if } f\, o_1 = (o_2, \vec{r}_2) \text{ and } f'\, o_2 = (o_3, \vec{r}_3) \\ \bot & \text{otherwise} \end{cases}$$

All memory operations are preserved by memory refinements. This property is not only useful for reasoning about program transformations, but also indicates that the memory interface does not expose internal details (such as variants of unions) that are unavailable in the memory of a (concrete) machine.

Lemma 7.7 *If* $m_1 \sqsubseteq^{f:\Delta_1\mapsto\Delta_2}_{\Gamma} m_2$ *and* $a_1 \sqsubseteq^{f:\Delta_1\mapsto\Delta_2}_{\Gamma} a_2 : \tau$ *and* $m_1\langle a_1\rangle_\Gamma = v_1$, *then there exists a value* v_2 *with* $m_2\langle a_2\rangle_\Gamma = v_2$ *and* $v_1 \sqsubseteq^{f:\Delta_1\mapsto\Delta_2}_{\Gamma} v_2 : \tau$.

Lemma 7.8 *If* $m_1 \sqsubseteq^{f:\Delta_1\mapsto\Delta_2}_{\Gamma} m_2$ *and* $a_1 \sqsubseteq^{f:\Delta_1\mapsto\Delta_2}_{\Gamma} a_2 : \tau$ *and* $v_1 \sqsubseteq^{f:\Delta_1\mapsto\Delta_2}_{\Gamma} v_2 : \tau$ *and* $\mathsf{writable}_\Gamma\, m_1\, a_1$, *then:*

1. *We have* $\mathsf{writable}_\Gamma\, m_2\, a_2$.
2. *We have* $m_1\langle a_1 := v_1\rangle_\Gamma \sqsubseteq^{f:\Delta_1\mapsto\Delta_2}_{\Gamma} m_2\langle a_2 := v_2\rangle_\Gamma$.

As shown in Lemma 6.63, storing a value v in memory and then retrieving it, does not necessarily yield the same value v. In case of a byte address, the value may have been stored in padding and therefore have become indeterminate. Secondly, it intentionally yields the value $|\, v\,|_\circ$ in which all pointers are frozen. However, the widely used compiler optimization of constant propagation, which substitutes values of known constants at compile time, is still valid in our memory model.

Lemma 7.9 *If* $\Gamma, \Delta \vdash v : \tau$, *then* $|\, v\,|_\circ \sqsubseteq^{\Delta}_{\Gamma} v : \tau$.

Theorem 7.10 (Constant propagation) *If* $\Gamma, \Delta \vdash m$ *and* $\Gamma, \Delta \vdash a : \tau$ *and* $\Gamma, \Delta \vdash v : \tau$ *and* $\mathsf{writable}_\Gamma\, a\, m$, *then there exists a value* v' *with:*

$$m\langle a := v\rangle_\Gamma\langle a\rangle_\Gamma = v' \quad \text{and} \quad v' \sqsubseteq^{\Delta}_{\Gamma} v : \tau.$$

Copying an object w by an assignment results in it being converted to a value $\mathsf{toval}_\Gamma\, w$ and back. This conversion makes invalid representations of base values indeterminate. Copying an object w byte-wise results in it being converted to bits $\overline{w}$ and back. This conversion makes all variants of unions unspecified. The following theorem shows that a copy by assignment can be transformed into a byte-wise copy.

Theorem 7.11 (Memcpy) *If* $\Gamma, \Delta \vdash w : \tau$*, then:*

$$\mathsf{ofval}_\Gamma\,(\overline{w}_{\mathbf{1}})\,(\mathsf{toval}_\Gamma\, w) \sqsubseteq^{\Delta}_{\Gamma} w \sqsubseteq^{\Delta}_{\Gamma} (\overline{w})^{\tau}_{\Gamma} : \tau.$$

Unused reads cannot be removed unconditionally in the $\mathrm{CH_2O}$ memory model because these have side-effects in the form of uses of the force_Γ operation that updates effective types. We show that uses of force_Γ can be removed for frozen addresses.

Theorem 7.12 *If* $\Gamma, \Delta \vdash m$ *and* $m\langle a\rangle_\Gamma \neq \bot$ *and* $\mathsf{frozen}\ a$*, then* $\mathsf{force}_\Gamma\, a\, m \sqsubseteq^{\Delta}_{\Gamma} m$.

7.3 Reasoning about disjointness

In order to prove soundness of the $\mathrm{CH_2O}$ axiomatic semantics, we often needed to reason about preservation of disjointness under memory operations [33]. This section describes some machinery to ease reasoning about disjointness. We show that our machinery, as originally developed in [31], extends to any separation algebra.

Definition 7.13 *Disjointness of a list* $\vec{x}$, notation $\bot\,\vec{x}$, is defined as:

1. $\bot\,\varepsilon$
2. If $\bot\,\vec{x}$ and $x \bot \bigcup \vec{x}$, then $\bot\,(x\,\vec{x})$

Notice that $\bot\,\vec{x}$ is stronger than having $x_i \bot x_j$ for each $i \neq j$. For example, using fractional permissions, we do not have $\bot\,[\,0.5,\,0.5,\,0.5\,]$ whereas $0.5 \bot 0.5$ clearly holds. Using disjointness of lists we can for example state the associativity law (law 3 of Definition 5.1) in a symmetric way:

Fact 7.14 If $\bot\,(x\ y\ z)$, then $x \cup (y \cup z) = (x \cup y) \cup z$.

We define a relation $\vec{x}_1 \equiv_\bot \vec{x}_2$ that expresses that $\vec{x}_1$ and $\vec{x}_2$ behave equivalently with respect to disjointness.

Definition 7.15 *Equivalence of lists* $\vec{x}_1$ *and* $\vec{x}_2$ *with respect to disjointness*, notation $\vec{x}_1 \equiv_\bot \vec{x}_2$, is defined as:

$$\vec{x}_1 \leq_\bot \vec{x}_2 := \forall x\,.\bot\,(x\,\vec{x}_1) \to \bot\,(x\,\vec{x}_2)$$
$$\vec{x}_1 \equiv_\bot \vec{x}_2 := \vec{x}_1 \leq_\bot \vec{x}_2 \wedge \vec{x}_2 \leq_\bot \vec{x}_1$$

It is straightforward to show that $\leq_\bot$ is reflexive and transitive, is respected by concatenation of lists, and is preserved by list containment. Hence, $\equiv_\bot$ is an equivalence relation, a congruence with respect to concatenation of lists, and is preserved by permutations. The following results (on arbitrary separation algebras) allow us to reason algebraically about disjointness.

Fact 7.16 If $\vec{x}_1 \leq_\bot \vec{x}_2$, then $\bot\,\vec{x}_1$ implies $\bot\,\vec{x}_2$.

Fact 7.17 If $\vec{x}_1 \equiv_\perp \vec{x}_2$, then $\perp\, \vec{x}_1$ iff $\perp\, \vec{x}_2$.

Theorem 7.18 *We have the following algebraic properties:*

$$\begin{aligned} \emptyset &\equiv_\perp \varepsilon && \\ x_1 \cup x_2 &\equiv_\perp x_1\, x_2 && \text{provided that } x_1 \perp x_2 \\ \bigcup \vec{x} &\equiv_\perp \vec{x} && \text{provided that } \perp\, \vec{x} \\ x_2 &\equiv_\perp x_1\, (x_2 \setminus x_1) && \text{provided that } x_1 \subseteq x_2 \end{aligned}$$

In Section 7.4 we show that we have similar properties as the above for the specific operations of our memory model.

7.4 The memory as a separation algebra

We show that the CH_2O memory model is a separation algebra, and that the separation algebra operations interact appropriately with the memory operations that we have defined in Section 6.

In order to define the separation algebra relations and operations on memories, we first define these on memory trees. Memory trees do not form a separation algebra themselves due to the absence of a unique $\emptyset$ element (memory trees have a distinct identity element new^τ_Γ for each type τ, see Definition 6.31). The separation algebra of memories is then defined by lifting the definitions on memory trees to memories (which are basically finite functions to memory trees).

Definition 7.19 The predicate $\mathsf{valid} : \mathsf{mtree} \to \mathsf{Prop}$ is inductively defined as:

$$\frac{\mathsf{valid}\ \vec{\mathbf{b}}}{\mathsf{valid}\ (\mathsf{base}_{\tau_\mathsf{b}}\ \vec{\mathbf{b}})} \qquad \frac{\mathsf{valid}\ \vec{w}}{\mathsf{valid}\ (\mathsf{array}_\tau\ \vec{w})} \qquad \frac{\mathsf{valid}\ \vec{w} \qquad \mathsf{valid}\ \vec{\vec{\mathbf{b}}}}{\mathsf{valid}\ (\mathsf{struct}_t\ \vec{w\vec{\mathbf{b}}})}$$

$$\frac{\mathsf{valid}\ w \qquad \mathsf{valid}\ \vec{\mathbf{b}} \qquad \neg\mathsf{unmapped}\ (\overline{w}\,\vec{\mathbf{b}})}{\mathsf{valid}\ (\mathsf{union}_t\ (i, w, \vec{\mathbf{b}}))} \qquad \frac{\mathsf{valid}\ \vec{\mathbf{b}}}{\mathsf{valid}\ (\overline{\mathsf{union}}_t\ \vec{\mathbf{b}})}$$

Fact 7.20 If $\Gamma, \Delta \vdash w : \tau$, then $\mathsf{valid}\ w$.

The valid predicate specifies the subset of memory trees on which the separation algebra structure is defined. The definition basically lifts the valid predicate from the leaves to the trees. The side-condition $\neg\mathsf{unmapped}\ (\overline{w}\,\vec{\mathbf{b}})$ on $\mathsf{union}_t\ (i, w, \vec{\mathbf{b}})$ memory trees ensures canonicity, unions whose permissions are unmapped cannot be accessed and are thus kept in unspecified variant. Unmapped unions $\overline{\mathsf{union}}_t\ \vec{\mathbf{b}}$ can be combined with other unions using $\cup$. The rationale for doing so will become clear in the context of the separation logic in the author's PhD thesis [33].

Definition 7.21 The relation $\bot : \mathsf{mtree} \to \mathsf{mtree} \to \mathsf{Prop}$ is inductively defined as:

$$\frac{\vec{\mathbf{b}}_1 \bot \vec{\mathbf{b}}_2}{\mathsf{base}_{\tau_\mathsf{b}}\,\vec{\mathbf{b}}_1 \bot \mathsf{base}_{\tau_\mathsf{b}}\,\vec{\mathbf{b}}_2} \qquad \frac{\vec{w}_1 \bot \vec{w}_2}{\mathsf{array}_\tau\,\vec{w}_1 \bot \mathsf{array}_\tau\,\vec{w}_2} \qquad \frac{\vec{w}_1 \bot \vec{w}_2 \qquad \overrightarrow{\vec{\mathbf{b}}_1} \bot \overrightarrow{\vec{\mathbf{b}}_2}}{\mathsf{struct}_t\,\overrightarrow{w_1\vec{\mathbf{b}}_1} \bot \mathsf{struct}_t\,\overrightarrow{w_2\vec{\mathbf{b}}_2}}$$

$$\frac{w_1 \bot w_2 \qquad \vec{\mathbf{b}}_1 \bot \vec{\mathbf{b}}_2 \qquad \neg\mathsf{unmapped}\,(\overline{w_1}\,\vec{\mathbf{b}}_1) \qquad \neg\mathsf{unmapped}\,(\overline{w_2}\,\vec{\mathbf{b}}_2)}{\mathsf{union}_t\,(i, w_1, \vec{\mathbf{b}}_1) \bot \mathsf{union}_t\,(i, w_2, \vec{\mathbf{b}}_2)}$$

$$\frac{\vec{\mathbf{b}}_1 \bot \vec{\mathbf{b}}_2}{\overline{\mathsf{union}_t}\,\vec{\mathbf{b}}_1 \bot \overline{\mathsf{union}_t}\,\vec{\mathbf{b}}_2} \qquad \frac{\overline{w_1}\,\vec{\mathbf{b}}_1 \bot \vec{\mathbf{b}}_2 \quad \mathsf{valid}\,w_1 \quad \neg\mathsf{unmapped}\,(\overline{w_1}\,\vec{\mathbf{b}}_1) \quad \mathsf{unmapped}\,\vec{\mathbf{b}}_2}{\mathsf{union}_t\,(i, w_1, \vec{\mathbf{b}}_1) \bot \overline{\mathsf{union}_t}\,\vec{\mathbf{b}}_2}$$

$$\frac{\vec{\mathbf{b}}_1 \bot \overline{w_2}\,\vec{\mathbf{b}}_2 \qquad \mathsf{valid}\,w_2 \qquad \mathsf{unmapped}\,\vec{\mathbf{b}}_1 \qquad \neg\mathsf{unmapped}\,(\overline{w_2}\,\vec{\mathbf{b}}_2)}{\overline{\mathsf{union}_t}\,\vec{\mathbf{b}}_1 \bot \mathsf{union}_t\,(i, w_2, \vec{\mathbf{b}}_2)}$$

Definition 7.22 The operation $\cup : \mathsf{mtree} \to \mathsf{mtree} \to \mathsf{mtree}$ is defined as:

$$\begin{aligned}
\mathsf{base}_{\tau_\mathsf{b}}\,\vec{\mathbf{b}}_1 \cup \mathsf{base}_{\tau_\mathsf{b}}\,\vec{\mathbf{b}}_2 &:= \mathsf{base}_{\tau_\mathsf{b}}\,(\vec{\mathbf{b}}_1 \cup \vec{\mathbf{b}}_2)\\
\mathsf{array}_\tau\,\vec{w}_1 \cup \mathsf{array}_\tau\,\vec{w}_2 &:= \mathsf{array}_\tau\,(\vec{w}_1 \cup \vec{w}_2)\\
\mathsf{struct}_t\,\overrightarrow{w_1\vec{\mathbf{b}}_1} \cup \mathsf{struct}_t\,\overrightarrow{w_2\vec{\mathbf{b}}_2} &:= \mathsf{struct}_t\,(\overrightarrow{w_1\vec{\mathbf{b}}_1} \cup \overrightarrow{w_2\vec{\mathbf{b}}_2})\\
\mathsf{union}_t\,(i, w_1, \vec{\mathbf{b}}_1) \cup \mathsf{union}_t\,(i, w_2, \vec{\mathbf{b}}_2) &:= \mathsf{union}_t\,(i, w_1 \cup w_2, \vec{\mathbf{b}}_1 \cup \vec{\mathbf{b}}_2)\\
\overline{\mathsf{union}_t}\,\vec{\mathbf{b}}_1 \cup \overline{\mathsf{union}_t}\,\vec{\mathbf{b}}_1 &:= \overline{\mathsf{union}_t}\,(\vec{\mathbf{b}}_1 \cup \vec{\mathbf{b}}_2)\\
\mathsf{union}_t\,(i, w_1, \vec{\mathbf{b}}_1) \cup \overline{\mathsf{union}_t}\,\vec{\mathbf{b}}_2 &:= \mathsf{union}_t\,(i, w_1, \vec{\mathbf{b}}_1)\,\hat{\cup}\,\vec{\mathbf{b}}_2\\
\overline{\mathsf{union}_t}\,\vec{\mathbf{b}}_1 \cup \mathsf{union}_t\,(i, w_2, \vec{\mathbf{b}}_2) &:= \mathsf{union}_t\,(i, w_2, \vec{\mathbf{b}}_2)\,\hat{\cup}\,\vec{\mathbf{b}}_1
\end{aligned}$$

In the last two clauses, $w\,\hat{\cup}\,\vec{\mathbf{b}}$ is a modified version of the memory tree w in which the elements on the leaves of w are zipped with $\vec{\mathbf{b}}$ using the $\cup$ operation on permission annotated bits (see Definitions 6.38 and 5.13).

The definitions of valid, $\bot$ and $\cup$ on memory trees satisfy all laws of a separation algebra (see Definition 5.1) apart from those involving $\emptyset$. We prove the cancellation law explicitly since it involves the aforementioned side-conditions on unions.

Lemma 7.23 *If $w_3 \bot w_1$ and $w_3 \bot w_2$ then:*

$$w_3 \cup w_1 = w_3 \cup w_2 \quad \text{implies} \quad w_1 = w_2.$$

Proof By induction on the derivations $w_3 \bot w_1$ and $w_3 \bot w_2$. We consider one case:

$$\frac{\mathsf{union}_t\,(i, w_3, \vec{\mathbf{b}}_3) \bot \mathsf{union}_t\,(i, w_1, \vec{\mathbf{b}}_1) \qquad \mathsf{union}_t\,(i, w_3, \vec{\mathbf{b}}_3) \bot \overline{\mathsf{union}_t}\,\vec{\mathbf{b}}_2 \qquad \mathsf{union}_t\,(i, w_3, \vec{\mathbf{b}}_3) \cup \mathsf{union}_t\,(i, w_1, \vec{\mathbf{b}}_1) = \mathsf{union}_t\,(i, w_3, \vec{\mathbf{b}}_3) \cup \overline{\mathsf{union}_t}\,\vec{\mathbf{b}}_2}{\mathsf{union}_t\,(i, w_1, \vec{\mathbf{b}}_1) = \overline{\mathsf{union}_t}\,\vec{\mathbf{b}}_2}$$

Here, we have $\overline{w_3}\,\vec{\mathbf{b}}_3 \cup \overline{w_1}\,\vec{\mathbf{b}}_1 = \overline{w_3}\,\vec{\mathbf{b}}_3 \cup \vec{\mathbf{b}}_2$ by assumption, and therefore $\overline{w_1}\,\vec{\mathbf{b}}_1 = \vec{\mathbf{b}}_2$ by the cancellation law of a separation algebra. However, by assumption we also have $\neg\mathsf{unmapped}\,(\overline{w_1}\,\vec{\mathbf{b}}_1)$ and $\mathsf{unmapped}\,\vec{\mathbf{b}}_2$, which contradicts $\overline{w_1}\,\vec{\mathbf{b}}_1 = \vec{\mathbf{b}}_2$.

Definition 7.24 The *separation algebra of memories* is defined as:

$$\begin{aligned}\mathsf{valid}\ m &:= \forall o\, w\, \mu\,.\, m\, o = (w,\ \mu) \to (\mathsf{valid}\ w \text{ and not } \overline{w} \text{ all } (\emptyset,\ \lightning))\\ m_1 \perp m_2 &:= \forall o\,.\, P\, m_1\, m_2\, o\\ m_1 \cup m_2 &:= \lambda o\,.\, f\, m_1\, m_2\, o\end{aligned}$$

$P : \mathsf{mem} \to \mathsf{mem} \to \mathsf{index} \to \mathsf{Prop}$ and $f : \mathsf{mem} \to \mathsf{mem} \to \mathsf{index} \to \mathsf{option\ mtree}$ are defined by case analysis on $m_1\, o$ and $m_2\, o$:

$m_1\, o$	$m_2\, o$	$P\, m_1\, m_2\, o$	$f\, m_1\, m_2\, o$
$(w_1,\ \mu)$	$(w_1,\ \mu)$	$w_1 \perp w_2$, not $\overline{w_1}$ all $(\emptyset,\ \lightning)$ and not $\overline{w_2}$ all $(\emptyset,\ \lightning)$	$(w_1 \cup w_2,\ \mu)$
$(w_1,\ \mu)$	$\perp$	valid w_1 and not $\overline{w_1}$ all $(\emptyset,\ \lightning)$	$(w_1,\ \mu)$
$\perp$	$(w_2,\ \mu)$	valid w_2 and not $\overline{w_2}$ all $(\emptyset,\ \lightning)$	$(w_2,\ \mu)$
τ_1	$\perp$	True	τ_1
$\perp$	τ_2	True	τ_2
$\perp$	$\perp$	True	$\perp$
otherwise		False	$\perp$

The definitions of the omitted relations and operations are as expected.

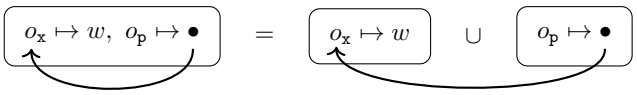
The emptiness conditions ensure canonicity. Objects that solely consist of indeterminate bits with $\emptyset$ permission are meaningless and should not be kept at all. These conditions are needed for cancellativity.

Fact 7.25 If $\Gamma, \Delta \vdash m$, then $\mathsf{valid}\ m$.

Lemma 7.26 *If $m_1 \perp m_2$, then:*

$$\Gamma, \Delta \vdash m_1 \cup m_2 \qquad \text{iff} \qquad \Gamma, \Delta \vdash m_1 \text{ and } \Gamma, \Delta \vdash m_2.$$

Notice that the memory typing environment Δ is not subdivided among m_1 and m_2. Consider the memory state corresponding to `int x = 10, *p = &x`:



Here, w is the memory tree that represents the integer value 10. The pointer on the right hand side is well-typed in the memory environment $\overline{o_{\mathtt{x}} \mapsto w,\ o_{\mathtt{p}} \mapsto \bullet}$ of the whole memory, but not in $\overline{o_{\mathtt{p}} \mapsto \bullet}$.

We prove some essential properties about the interaction between the separation algebra operations and the memory operations. These properties have been used in the soundness proof of the separation logic in the author's PhD thesis [33].

Lemma 7.27 (Preservation of lookups) *If $\Gamma, \Delta \vdash m_1$ and $m_1 \subseteq m_2$, then:*

$$\begin{aligned} m_1\langle a\rangle_\Gamma = v \quad &\text{implies} \quad m_2\langle a\rangle_\Gamma = v\\ \mathsf{writable}_\Gamma\, a\, m_1 \quad &\text{implies} \quad \mathsf{writable}_\Gamma\, a\, m_2\end{aligned}$$

The relation $\subseteq$ is part of a separation algebra, see Definition 5.1. We have $m_1 \subseteq m_2$ iff there is an m_3 with $m_1 \perp m_3$ and $m_2 = m_1 \cup m_3$.

Lemma 7.28 (Preservation of disjointness) *If* $\Gamma, \Delta \vdash m$ *then:*

$$\begin{aligned}
&m \leq_\perp \mathsf{force}_\Gamma\, a\, m && \text{if } \Gamma, \Delta \vdash a : \tau \text{ and } m\langle a\rangle_\Gamma \neq \perp\\
&m \leq_\perp m\langle a := v\rangle_\Gamma && \text{if } \Gamma, \Delta \vdash a : \tau \text{ and } \mathsf{writable}_\Gamma\, a\, m\\
&m \leq_\perp \mathsf{lock}_\Gamma\, a\, m && \text{if } \Gamma, \Delta \vdash a : \tau \text{ and } \mathsf{writable}_\Gamma\, a\, m\\
&m \leq_\perp \mathsf{unlock}\ \Omega\ m && \text{if } \Omega \subseteq \mathsf{locks}\ m
\end{aligned}$$

The relation $\leq_\perp$ *is defined in Definition 7.15. If* $m \leq_\perp m'$*, then each memory that is disjoint to* m *is also disjoint to* m'*.*

As a corollary of the above lemma and Fact 7.16 we obtain that $m_1 \perp m_2$ implies disjointness of the memory operations:

$$\begin{aligned}
&\mathsf{force}_\Gamma\, a\, m_1 \perp m_2 && m_1\langle a := v\rangle_\Gamma \perp m_2\\
&\mathsf{lock}_\Gamma\, a\, m_1 \perp m_2 && \mathsf{unlock}\ \Omega\ m_1 \perp m_2
\end{aligned}$$

Lemma 7.29 (Unions distribute) *If* $\Gamma, \Delta \vdash m$ *and* $m_1 \perp m_2$ *then:*

$$\begin{aligned}
\mathsf{force}_\Gamma\, a\, (m_1 \cup m_2) &= \mathsf{force}_\Gamma\, a\, m_1 \cup m_2 && \text{if } \Gamma, \Delta \vdash a : \tau \text{ and } m_1\langle a\rangle_\Gamma \neq \perp\\
(m_1 \cup m_2)\langle a := v\rangle_\Gamma &= m_1\langle a := v\rangle_\Gamma \cup m_2 && \text{if } \Gamma, \Delta \vdash \{a, v\} : \tau \text{ and } \mathsf{writable}_\Gamma\, a\, m_1\\
\mathsf{lock}_\Gamma\, a\, (m_1 \cup m_2) &= \mathsf{lock}_\Gamma\, a\, m_1 \cup m_2 && \text{if } \Gamma, \Delta \vdash a : \tau \text{ and } \mathsf{writable}_\Gamma\, a\, m_1\\
\mathsf{unlock}\ \Omega\ (m_1 \cup m_2) &= \mathsf{unlock}\ \Omega\ m_1 \cup m_2 && \text{if } \Omega \subseteq \mathsf{locks}\ m_1
\end{aligned}$$

Memory trees and memories can be generalized to contain elements of an arbitrary separation algebra as leaves instead of just permission annotated bits [32]. These generalized memories form a functor that lifts the separation algebra structure on the leaves to entire trees. We have taken this approach in the Coq development, but for brevity's sake, we have refrained from doing so in this paper.

8 Formalization in Coq

Real-world programming language have a large number of features that require large formal descriptions. As this paper has shown, the C programming language is not different in this regard. On top of that, the C semantics is very subtle due to an abundance of delicate corner cases. Designing a semantics for C and proving properties about such a semantics therefore inevitably requires computer support.

For these reasons, we have used Coq [15] to formalize all results in this paper. Although Coq does not guarantee the absence of mistakes in our definitions, it provides a rigorous set of checks on our definitions, for example by its type checking of definitions. On top of that, we have used Coq to prove all metatheoretical results stated in this paper. Last but not least, using Coq's program extraction facility we have extracted an exploration tool to test our memory model on small example programs [33, 37]. Despite our choice to use Coq, we believe that nearly all parts of CH_2O could be formalized in any proof assistant based on higher-order logic.

8.1 Overloaded typing judgments

Type classes are used to overload notations for typing judgments (we have 25 different typing judgments). The class `Valid` is used for judgments without a type, such as $\vdash \Gamma$ and $\Gamma, \Delta \vdash m$.

```
Class Valid (E A : Type) := valid: E → A → Prop.
Notation "✓{ Γ }" := (valid Γ).
Notation "✓{ Γ }*" := (Forall (✓{Γ})).
```

We use product types to represent judgments with multiple environments such as $\Gamma, \Delta \vdash m$. The notation `✓{Γ}*` is used to lift the judgment to lists. The class `Typed` is used for judgments such as $\Gamma, \Delta \vdash v : \tau$ and $\Gamma, \Delta, \vec{\tau} \vdash e : \tau_{\mathsf{lr}}$.

```
Class Typed (E T V : Type) := typed: E → V → T → Prop.
Notation "Γ ⊢ v : τ" := (typed Γ v τ).
Notation "Γ ⊢* vs :* τs" := (Forall2 (typed Γ) vs τs).
```

8.2 Implementation-defined behavior

Type classes are used to parameterize the whole Coq development by implementation-defined parameters such as integer sizes. For example, Lemma 6.51 looks like:

```
Lemma to_of_val `{EnvSpec K} Γ Δ γs v τ :
  ✓ Γ → (Γ,Δ) ⊢ v : τ → length γs = bit_size_of Γ τ →
  to_val Γ (of_val Γ γs v) = freeze true v.
```

The parameter `EnvSpec` K is a type class describing an implementation environment with ranks K (Definition 4.12). Just as in this paper, the type K of integer ranks is a parameter of the inductive definition of types (see Definition 4.1) and is propagated through all syntax.

```
Inductive signedness := Signed | Unsigned.
Inductive int_type (K: Set) := IntType {sign: signedness; rank: K}.
```

The definition of the type class `EnvSpec` is based on the approach of Spitters and van der Weegen [55]. We have a separate class `Env` for the operations that is an implicit parameter of the whole class and all lemmas.

```
Class Env (K: Set) := {
  env_type_env :> IntEnv K;
  size_of : env K → type K → nat;
  align_of : env K → type K → nat;
  field_sizes : env K → list (type K) → list nat
}.
Class EnvSpec (K: Set) `{Env K} := {
  int_env_spec :>> IntEnvSpec K;
  size_of_ptr_ne_0 Γ τp : size_of Γ (τp.*) ≠ 0;
  size_of_int Γ τi : size_of Γ (intT τi) = rank_size (rank τi);
  ...
}.
```

8.3 Partial functions

Although many operations in CH_2O are partial, we have formalized many such operations as total functions that assign an appropriate default value. We followed the approach presented in Section 5.2 where operations are combined with a *validity predicate* that describes in which case they may be used. For example, part (2) of Lemma 7.29 is stated in the Coq development as follows:

```
Lemma mem_insert_union `{EnvSpec K} Γ Δ m1 m2 a1 v1 τ1 :
  ✓ Γ → ✓{Γ,Δ} m1 → m1 ⊥ m2 →
  (Γ,Δ) ⊢ a1 : TType τ1 → mem_writable Γ a1 m1 → (Γ,Δ) ⊢ v1 : τ1
      →
  <[a1:=v1]{Γ}>(m1 ∪ m2) = <[a1:=v1]{Γ}>m1 ∪ m2.
```

Here, `m1 ⊥ m2` is the side-condition of `m1 ∪ m2`, and `mem_writable Γ a1 m1` the side-condition of `<[a1:=v1]{Γ}>m1`. Alternatives approaches include using the option monad or dependent types, but our approach proved more convenient. In particular, since most validity predicates are given by an inductive definition, various proofs could be done by induction on the structure of the validity predicate. The cases one has to consider correspond exactly to the domain of the partial function.

Admissible side-conditions, such as in the above example `<[a1:=v1]{Γ}>m1 ⊥ m2` and `mem_writable Γ a1 (m1 ∪ m2)`, do not have to be stated explicitly and follow from the side-conditions that are already there. By avoiding the need to state admissible side-conditions, we avoid a blow-up in the number of side-conditions of many lemmas. We thus reduce the proof effort needed to use such a lemma.

8.4 Automation

The proof style deployed in the CH_2O development combines interactive proofs with automated proofs. In this section we describe some tactics and forms of proof automation deployed in the CH_2O development.

Small inversions. Coq's `inversion` tactic has two serious shortcomings on inductively defined predicates with many constructors. It is rather slow and its way of controlling of names for variables and hypotheses is deficient. Hence, we often used the technique of small inversions by Monin and Shi [43] that improves on both shortcomings.

Solving disjointness. We have used Coq's setoid machinery [54] to enable rewriting using the relations $\leq_\perp$ and $\equiv_\perp$ (Definition 7.15). Using this machinery, we have implemented a tactic that automatically solves entailments of the form:

$$H_0 : \perp \vec{x}_0,\ \ldots,\ H_n : \perp \vec{x}_{n-1} \quad \vdash \quad \perp \vec{x}$$

where $\vec{x}$ and $\vec{x}_i$ (for $i < n$) are arbitrary Coq expressions built from $\emptyset$, $\cup$ and $\bigcup$. This tactic works roughly as follows:

1. Simplify hypotheses using Theorem 7.18.
2. Solve side-conditions by simplification using Theorem 7.18 and a solver for list containment (implemented by reflection).

3. Repeat these steps until no further simplification is possible.
4. Finally, solve the goal by simplification using Theorem 7.18 and list containment.

This tactic is not implemented using reflection, but that is something we intend to do in future work to improve its performance.

First-order logic. Many side-conditions we have encountered involve simple entailments of first-order logic such as distributing logical quantifiers combined with some propositional reasoning. Coq does not provide a solver for first-order logic apart from the `firstorder` tactic whose performance is already insufficient on small goals.

We have used Ltac to implemented an ad-hoc solver called `naive_solver`, which performs a simple breath-first search proof search. Although this tactic is inherently incomplete and suffers from some limitations, it turned out to be sufficient to solve many uninteresting side-conditions (without the need for classical axioms).

8.5 Overview of the Coq development

The Coq development of the memory model, which is entirely constructive and axiom free, consists of the following parts:

Component	**Sections**	**LOC**
Support library (lists, finite sets, finite maps, *etc.*)	Section 2	12 524
Types & Integers	Section 4	1 928
Permissions & separation algebras	Section 5	1 811
Memory model	Section 6	8 736
Refinements	Section 7.2	4 046
Memory as separation algebra	Section 7.4	3 844
Total		32 889

9 Related work

The idea of using a memory model based on trees instead of arrays of plain bits, and the idea of using pointers based on paths instead of offsets, has already been used for object oriented languages. It goes back at least to Rossie and Friedman [51], and has been used by Ramananandro *et al.* [48] for C++. Furthermore, many researchers have considered connections between unstructured and structured views of data in C [2, 14, 21, 56] in the context of program logics.

However, a memory model that combines an abstract tree based structure with low-level object representations in terms of bytes has not been explored before. In this section we will describe other formalizations of the C memory model.

Norrish (1998) Norrish has formalized a significant fragment of the C89 standard using the proof assistant HOL4 [44, 45]. He was the first to describe non-determinism and sequence points formally. Our treatment of these features has partly been based on his work. Norrish's formalization of the C type system has some similarities with our type system: he has also omitted features that can be desugared and has proven type preservation.

Contrary to our work, Norrish has used an unstructured memory model based on sequences of bytes. Since he has considered the C89 standard in which effective types (and similar notions) were not introduced yet, his choice is appropriate. For C99 and beyond, a more detailed memory model like ours is needed, see also Section 3 and Defect Report #260 and #451 [26].

Another interesting difference is that Norrish represents abstract values (integers, pointers and structs) as sequences of bytes instead of mathematical values. Due to this, padding bytes retain their value while structs are copied. This is not faithful to the C99 standard and beyond.

Leroy et al. (2006) Leroy *et al.* have formalized a significant part of C using the Coq proof assistant [38, 39]. Their part of C, which is called CompCert C, covers most major features of C and can be compiled into assembly (PowerPC, ARM and x86) using a compiler written in Coq. Their compiler, called CompCert, has been proven correct with respect to the CompCert C and assembly semantics.

The goal of CompCert is essentially different from CH_2O's. What can be proven with respect to the CompCert semantics does not have to hold for *any* C11 compiler, it just has to hold for the CompCert compiler. CompCert is therefore in its semantics allowed to restrict implementation defined behaviors to be very specific (for example, it uses 32-bit `int`s since it targets only 32-bit computing architectures) and allowed to give a defined semantics to various undefined behaviors (such as sequence point violations, violations of effective types, and certain uses of dangling pointers).

The CompCert memory model is used by all languages (from C until assembly) of the CompCert compiler [40, 41]. The CompCert memory is a finite partial function from object identifiers to objects. Each local, global and static variable, and invocation of `malloc` is associated with a unique object identifier of a separate object in memory. We have used the same approach in CH_2O, but there are some important differences. The paragraphs below discuss the relation of CH_2O with the first and second version of the CompCert memory model.

Leroy and Blazy (2008) In the first version of the CompCert memory model [41], objects were represented as arrays of type-annotated fragments of base values. Examples of bytes are thus "the 2nd byte of the short 13" or "the 3rd byte of the pointer (o, i)". Pointers were represented as pairs (o, i) where o is an object identifier and i the byte offset into the object o.

Since bytes are annotated with types and could only be retrieved from memory using an expression of matching type, effective types on the level of base types are implicitly described. However, this does not match the C11 standard. For example, Leroy and Blazy do assign the return value 11 to the following program:

```
struct S1 { int x; };
struct S2 { int y; };
int f(struct S1 *p, struct S2 *q) {
  p->x = 10;
  q->y = 11;
  return p->x;
}
int main() {
  union U { struct S1 s1; struct S2 s2; } u;
```

```
  printf("%d\n", f(&u.s1, &u.s2));
}
```

This code strongly resembles example [27, 6.5.2.3p9] from the C11 standard, which is stated to have undefined behavior[6]. GCC and Clang optimize this code to print 10, which differs from the value assigned by Leroy and Blazy.

Apart from assigning too much defined behavior, Leroy and Blazy's treatment of effective types also prohibits any form of "bit twiddling".

Leroy and Blazy have introduced the notion of memory injections in [41]. This notion allows one to reason about memory transformations in an elegant way. Our notion of memory refinements (Section 7.2) generalize the approach of Leroy and Blazy to a tree based memory model.

Leroy et al. (2012) The second version of CompCert memory model [40] is entirely untyped and is extended with permissions. Symbolic bytes are only used for pointer values and indeterminate storage, whereas integer and floating point values are represented as numerical bytes (integers between 0 and $2^8 - 1$).

We have extended this approach by analogy to bit-representations, representing indeterminate storage and pointer values using symbolic bits, and integer values using concrete bits. This choice is detailed in Section 6.2.

As an extension of CompCert, Robert and Leroy have formally proven soundness of an alias analysis [50]. Their alias analysis is untyped and operates on the RTL intermediate language of CompCert.

Beringer *et al.* [7] have developed an extension of CompCert's memory injections to reason about program transformations in the case of separate compilation. The issues of separate compilation are orthogonal to those that we consider.

Appel et al. (2014) The Verified Software Toolchain (VST) by Appel *et al.* provides a higher-order separation logic for Verifiable C, which is a variant of CompCert's intermediate language Clight [3].

The VST is intended to be used together with the CompCert compiler. It gives very strong guarantees when done so. The soundness proof of the VST in conjunction with the correctness proof of the CompCert compiler ensure that the proven properties also hold for the generated assembly.

In case the verified program is compiled with a compiler different from CompCert, the trust in the program is still increased, but no full guarantees can be given. That is caused by the fact that CompCert's intermediate language Clight uses a specific evaluation order and assigns defined behavior to many undefined behaviors of the C11 standard. For example, Clight assigns defined behavior to violations of effective types and sequence point violations. The VST inherits these defined behaviors from CompCert and allows one to use them in proofs.

Since the VST is linked to CompCert, it uses CompCert's coarse permission system on the level of the operational semantics. Stewart and Appel [3, Chapter 42] have introduced a way to use a more fine grained permission system at the level of the separation logic without having to modify the Clight operational semantics. Their approach shows its merits when used for concurrency, in which case the memory model contains *ghost* data related to the conditions of locks [23, 24].

[6] We have modified the example from the standard slightly in order to trigger optimizations by GCC and Clang.

Besson et al. (2014) Besson *et al.* have proposed an extension of the CompCert memory model that assigns a defined semantics to operations that rely on the numerical values of uninitialized memory and pointers [8].

Objects in their memory model consist of lazily evaluated values described by symbolic expressions. These symbolic expressions are used to delay the evaluation of operations on uninitialized memory and pointer values. Only when a concrete value is needed (for example in case of the controlling expression of an **if-then-else**, **for**, or **while** statement), the symbolic expression is normalized. Consider:

```
int x, *p = &x;
int y = ((unsigned char*)p)[1] | 1;
// y has symbolic value "2nd pointer byte of p" | 1
if (y & 1) printf("one\n"); // unique normalization -> OK
if (y & 2) printf("two\n"); // no unique normalization -> bad
```

The value of `((unsigned char*)p)[1] | 1` is not evaluated eagerly. Instead, the assignment to `y` stores a symbolic expression denoting this value. During the execution of the first **if** statement, the actual value of `y & 1` is needed. In this case, `y & 1` has the value 1 for any possible numerical value of `((unsigned char*)p)[1]`. As a result, the string `one` is printed.

The semantics of Besson *et al.* is deterministic by definition. Normalization of symbolic expressions has defined behavior if and only if the expression can be normalized to a unique value under any choice of numeral values for pointer representations and uninitialized storage. In the second **if** statement this is not the case.

The approach of Besson *et al.* gives a semantics to some programming techniques that rely on the numerical representations of pointers and uninitialized memory. For example, it gives an appropriate semantics to pointer tagging in which unused bits of a pointer representation are used to store additional information.

However, as already observed by Kang *et al.* [28], Besson *et al.* do not give a semantics to many other useful cases. For example, printing the object representation of a struct, or computing the hash of a pointer value, is inherently non-deterministic. The approach of Besson *et al.* assigns undefined behavior to these use cases.

The goal of Besson *et al.* is inherently different from ours. Our goal is to describe the C11 standard faithfully whereas Besson *et al.* focus on *de facto* versions of C. They intentionally assign defined behavior to many constructs involving uninitialized memory that are clearly undefined according to the C11 standard, but that are nonetheless faithfully compiled by specific compilers.

Ellison and Roşu (2012) Ellison and Roşu [18, 19] have developed an executable semantics of the C11 standard using the $\mathbb{K}$-framework[7]. Their semantics is very comprehensive and describes all features of a freestanding C implementation [27, 4p6] including some parts of the standard library. It furthermore has been thoroughly tested against test suites (such as the GCC torture test suite), and has been used as an oracle for compiler testing [49].

Ellison and Roşu support more C features than we do, but they do not have infrastructure for formal proofs, and thus have not established any metatheoretical properties about their semantics. Their semantics, despite being written in a formal framework, should more be seen as a debugger, a state space search tool, or possibly,

[7] This work has been superseded by Hathhorn *et al.* [22], which is described below.

as a model checker. It is unlikely to be of practical use in proof assistants because it is defined on top of a large C abstract syntax and uses a rather ad-hoc execution state that contains over 90 components.

Similar to our work, Ellison and Roşu's goal is to *exactly* describe the C11 standard. However, for some programs their semantics is less precise than ours, which is mainly caused by their memory model, which is less principled than ours. Their memory model is based on CompCert's: it is essentially a finite map of objects consisting of unstructured arrays of bytes.

Hathhorn et al. (2015) Hathhorn *et al.* [22] have extended the work of Ellison and Roşu to handle more underspecification of C11. Most importantly, the memory model has been extended and support for the type qualifiers `const`, `restrict` and `volatile` has been added.

Hathhorn *et al.* have extended the original memory model (which was based on CompCert's) with decorations to handle effective types, restrictions on padding and the `restrict` qualifier. Effective types are modeled by a map that associates a type to each object. Their approach is less fine-grained than ours and is unable to account for active variants of unions. It thus does not assign undefined behavior to important violations of effective types and in turn does not allow compilers to perform optimizations based on type-based alias analysis. For example:

```
// Undefined behavior in case f is called with aliased
// pointers due to effective types
int f(short *p, int *q) { *p = 10; *q = 11; return *p; }
int main() {
  union { short x; int y; } u = { .y = 0 };
  return f(&u.x, &u.y);
}
```

The above program has undefined behavior due to a violation of effective types. This is captured by our tree based memory model, but Hathhorn *et al.* require the program to return the value 11. When compiled with GCC or Clang with optimization level `-O2`, the compiled program returns the value 10.

Hathhorn *et al.* handle restrictions on padding bytes in the case of unions, but not in the case of structs. For example, the following program returns the value 1 according to their semantics, whereas it has unspecified behavior according to the C11 standard [27, 6.2.6.1p6] (see also Section 3.2):

```
struct S { char a; int b; } s;
((unsigned char*)(&s))[1] = 1;
s.a = 10; // Makes the padding bytes of 's' indeterminate
return ((unsigned char*)(&s))[1];
```

The restrictions on paddings bytes are implicit in our memory model based on structured trees, and thus handled correctly. The above examples provide evidence that a structured approach, especially combined with metatheoretical results, is more reliable than depending on ad-hoc decorations.

Kang et al. (2015) Kang *et al.* [28] have proposed a memory model that gives a semantics to pointer to integer casts. Their memory model uses a combination of

numerical and symbolic representations of pointer values (whereas CompCert and CH_2O always represent pointer values symbolically). Initially each pointer is represented symbolically, but whenever the numerical representation of a pointer is needed (due to a pointer to integer cast), it is non-deterministically *realized*.

The memory model of Kang *et al.* gives a semantics to pointer to integer casts while allowing common compiler optimizations that are invalid in a naive low-level memory model. They provide the following motivating example:

```
void g(void) { ... }
int f(void) {
  int a = 0;
  g();
  return a;
}
```

In a concrete memory model, there is the possibility that the function `g` is able to *guess* the numerical representation of `&a`, and thereby access or even modify `a`. This is undesirable, because it prevents the widely used optimization of constant propagation, which optimizes the variable `a` out.

In the CompCert and CH_2O memory model, where pointers are represented symbolically, it is guaranteed that `f` has exclusive control over `a`. Since `&a` has not been leaked, `g` can impossibly access `a`. In the memory model of Kang *et al.* a pointer will only be given a numerical representation when it is cast to an integer. In the above code, no such casts appear, and `g` cannot access `a`.

The goal of Kang *et al.* is to give a unambiguous mathematical model for pointer to integer casts, but not necessarily to comply with C11 or existing compilers. Although we think that their model is a reasonable choice, it is unclear whether it is faithful to the C11 standard in the context of Defect Report #260 [26]. Consider:

```
int x = 0, *p = 0;
for (uintptr_t i = 0; ; i++) {
  if (i == (uintptr_t)&x) {
    p = (int*)i; break;
  }
}
*p = 15;
printf("%d\n", x);
```

Here we loop through the range of integers of type `uintptr_t` until we have found the integer representation `i` of `&x`, which we then assign to the pointer `p`.

When compiled with `gcc -O2` (version 4.9.2), the generated assembly no longer contains a loop, and the pointers `p` and `&x` are assumed not to alias. As a result, the program prints the old value of `x`, namely `0`. In the memory model of Kang *et al.* the pointer obtained via the cast `(int*)i` is exactly the same as `&x`. In their model the program thus has defined behavior and is required to print `15`.

We have reported this issue to the GCC bug tracker[8]. However it unclear whether the GCC developers consider this a bug or not. Some developers seem to believe that this program has undefined behavior and that GCC's optimizations are thus justified. Note that the cast `(intptr_t)&x` is already forbidden by the type system of CH_2O.

[8] See `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752`.

10 Conclusion

In this paper we have given a formal description of a significant part of the non-concurrent C11 memory model. This formal description has been used in [33, 37] as part of an an operational, executable and axiomatic semantics of C. On top of this formal description, we have provided a comprehensive collection of metatheoretical results. All of these results have been formalized using the Coq proof assistant.

It would be interesting to investigate whether our memory model can be used to help the standard committee to improve future versions of the standard. For example, whether it could help to improve the standard's prose description of effective types. As indicated on page 4 of Section 1, the standard's description is not only ambiguous, but also does not cover its intent to enable type-based alias analysis. The description of our memory model is unambiguous and allows one to express intended consequences formally. We have formally proven soundness of an abstract version of type-based alias analysis with respect to our memory model (Theorem 7.2).

An obvious direction for future work is to extend the memory model with additional features. We give an overview of some features of C11 that are absent.

- **Floating point arithmetic.** Representations of floating point numbers and the behaviors of floating point arithmetic are subject to a considerable amount of implementation defined behavior [27, 5.2.4.2.2].
  First of all, one could restrict to IEEE-754 floating point arithmetic, which has a clear specification [25] and a comprehensive formalization in Coq [10]. Boldo *et al.* have taken this approach in the context of CompCert [9] and we see no fundamental problems applying it to $\mathrm{CH_2O}$ as well.
  Alternatively, one could consider formalizing all implementation-defined aspects of the description of floating arithmetic in the C11 standard.
- **Bitfields.** Bitfields are fields of struct types that occupy individual bits [27, 6.7.2.1p9]. We do not foresee fundamental problems adding bitfields to $\mathrm{CH_2O}$ as bits already constitute the smallest unit of storage in our memory model.
- **Untyped malloc.** $\mathrm{CH_2O}$ supports dynamic memory allocation via an operator $\mathsf{alloc}_\tau\ e$ close to C++'s `new` operator. The $\mathsf{alloc}_\tau\ e$ operator yields a $\tau*$ pointer to storage for a τ-array of length e. This is different from C's `malloc` function that yields a `void*` pointer to storage of unknown type [27, 7.22.3.4].
  Dynamic memory allocation via the untyped `malloc` function is closely related to unions and effective types. Only when dynamically allocated storage is actually used, it will receive an effective type. We expect one could treat `malloc`ed objects as unions that range over all possible types that fit.
- **Restrict qualifiers.** The `restrict` qualifier can be applied to any pointer type to express that the pointers do not alias. Since the description in the C11 standard [27, 6.7.3.1] is ambiguous (most notably, it is unclear how it interacts with nested pointers and data types), formalization and metatheoretical proofs may provide prospects for clarification.
- **Volatile qualifiers.** The `volatile` qualifier can be applied to any type to indicate that its value may be changed by an external process. It is meant to prevent compilers from optimizing away data accesses or reordering these [27, footnote 134]. Volatile accesses should thus be considered as a form of I/O.
- **Concurrency and atomics.** Shared-memory concurrency and atomic operations are the main omission from the C11 standard in the $\mathrm{CH_2O}$ semantics.

Although shared-memory concurrency is a relatively new addition to the C and C++ standards, there is already a large body of ongoing work in this direction, see for example [4, 5, 52, 53, 57]. These works have led to improvements of the standard text.

There are still important open problems in the area of concurrent memory models for already small sublanguages of C [4]. Current memory models for these sublanguages involve just features specific to threads and atomic operations whereas we have focused on structs, unions, effective types and indeterminate memory. We hope that both directions are largely orthogonal and will eventually merge into a fully fledged C11 memory model and semantics.

Acknowledgments. I thank my supervisors Freek Wiedijk and Herman Geuvers for their helpful suggestions. I thank Xavier Leroy, Andrew Appel, Lennart Beringer and Gordon Stewart for many discussions on the CompCert memory model, and the anonymous reviewers for their feedback. This work is financed by the Netherlands Organisation for Scientific Research (NWO), project 612.001.014.

References

1. R. Affeldt and N. Marti. Towards formal verification of TLS network packet processing written in C. In *PLPV*, pages 35–46, 2013.
2. R. Affeldt and K. Sakaguchi. An Intrinsic Encoding of a Subset of C and its Application to TLS Network Packet Processing. *JFR*, 7(1), 2014.
3. A. W. Appel, editor. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.
4. M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell. The Problem of Programming Language Concurrency Semantics. In *ESOP*, volume 9032 of *LNCS*, pages 283–307, 2015.
5. M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, pages 55–66, 2011.
6. J. Bengtson, J. B. Jensen, F. Sieczkowski, and L. Birkedal. Verifying Object-Oriented Programs with Higher-Order Separation Logic in Coq. In *ITP*, volume 6898 of *LNCS*, pages 22–38, 2011.
7. L. Beringer, G. Stewart, R. Dockins, and A. W. Appel. Verified Compilation for Shared-Memory C. In *ESOP*, volume 8410 of *LNCS*, pages 107–127, 2014.
8. F. Besson, S. Blazy, and P. Wilke. A Precise and Abstract Memory Model for C Using Symbolic Values. In *APLAS*, volume 8858 of *LNCS*, pages 449–468, 2014.
9. S. Boldo, J.-H. Jourdan, X. Leroy, and G. Melquiond. A Formally-Verified C Compiler Supporting Floating-Point Arithmetic. In *ARITH*, pages 107–115, 2013.
10. S. Boldo and G. Melquiond. Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In *ARITH*, pages 243–252, 2011.
11. R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission Accounting in Separation Logic. In *POPL*, pages 259–270, 2005.
12. J. Boyland. Checking Interference with Fractional Permissions. In *SAS*, volume 2694 of *LNCS*, pages 55–72, 2003.
13. C. Calcagno, P. W. O'Hearn, and H. Yang. Local Action and Abstract Separation Logic. In *LICS*, pages 366–378, 2007.

14. E. Cohen, M. Moskal, S. Tobies, and W. Schulte. A Precise Yet Efficient Memory Model For C. *ENTCS*, 254:85–103, 2009.
15. Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2015. Available at `https://coq.inria.fr/doc/`.
16. E. W. Dijkstra. Cooperating sequential processes. In *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.
17. R. Dockins, A. Hobor, and A. W. Appel. A Fresh Look at Separation Algebras and Share Accounting. In *APLAS*, volume 5904 of *LNCS*, pages 161–177, 2009.
18. C. Ellison. *A Formal Semantics of C with Applications*. PhD thesis, University of Illinois, 2012.
19. C. Ellison and G. Roşu. An executable formal semantics of C with applications. In *POPL*, pages 533–544, 2012.
20. GCC. The GNU Compiler Collection. Website, available at `http://gcc.gnu.org/`.
21. D. Greenaway, J. Lim, J. Andronick, and G. Klein. Don't Sweat the Small Stuff: Formal Verification of C Code Without the Pain. In *PLDI*, pages 429–439, 2014.
22. C. Hathhorn, C. Ellison, and G. Roşu. Defining the Undefinedness of C. In *PLDI*, pages 336–345, 2015.
23. A. Hobor. *Oracle Semantics*. PhD thesis, Princeton University, 2008.
24. A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle Semantics for Concurrent Separation Logic. In *ESOP*, volume 4960 of *LNCS*, pages 353–367, 2008.
25. IEEE Computer Society. *754-2008: IEEE Standard for Floating Point Arithmetic*. IEEE, 2008.
26. ISO. WG14 Defect Report Summary. Website, available at `http://www.open-std.org/jtc1/sc22/wg14/www/docs/`.
27. ISO. *ISO/IEC 9899-2011: Programming languages – C*. ISO Working Group 14, 2012.
28. J. Kang, C.-K. Hur, W. Mansky, D. Garbuzov, S. Zdancewic, and V. Vafeiadis. A Formal C Memory Model Supporting Integer-Pointer Casts. In *PLDI*, pages 326–335, 2015.
29. G. Klein, R. Kolanski, and A. Boyton. Mechanised Separation Algebra. In *ITP*, volume 7406 of *LNCS*, pages 332–337, 2012.
30. R. Krebbers. Aliasing restrictions of C11 formalized in Coq. In *CPP*, volume 8307 of *LNCS*, 2013.
31. R. Krebbers. An Operational and Axiomatic Semantics for Non-determinism and Sequence Points in C. In *POPL*, pages 101–112, 2014.
32. R. Krebbers. Separation algebras for C verification in Coq. In *VSTTE*, volume 8471 of *LNCS*, pages 150–166, 2014.
33. R. Krebbers. *The C standard formalized in Coq*. PhD thesis, Radboud University, 2015.
34. R. Krebbers, X. Leroy, and F. Wiedijk. Formal C semantics: CompCert and the C standard. In *ITP*, volume 8558 of *LNCS*, pages 543–548, 2014.
35. R. Krebbers and F. Wiedijk. A Formalization of the C99 Standard in HOL, Isabelle and Coq. In *CICM*, volume 6824 of *LNCS*, pages 297–299, 2011.
36. R. Krebbers and F. Wiedijk. Separation Logic for Non-local Control Flow and Block Scope Variables. In *FoSSaCS*, volume 7794 of *LNCS*, pages 257–272, 2013.
37. R. Krebbers and F. Wiedijk. A Typed C11 Semantics for Interactive Theorem Proving. In *CPP*, pages 15–27, 2015.

38. X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*, pages 42–54, 2006.
39. X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.
40. X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. The CompCert Memory Model, Version 2. Research report RR-7987, INRIA, 2012. Revised version available as Chapter 32 of [3].
41. X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *JAR*, 41(1):1–31, 2008.
42. N. Maclaren. What is an Object in C Terms?, 2001. Mailing list message, available at `http://www.open-std.org/jtc1/sc22/wg14/9350`.
43. J. Monin and X. Shi. Handcrafted Inversions Made Operational on Operational Semantics. In *ITP*, volume 7998 of *LNCS*, pages 338–353, 2013.
44. M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998.
45. M. Norrish. Deterministic Expressions in C. In *ESOP*, volume 1576 of *LNCS*, pages 147–161, 1999.
46. P. W. O'Hearn. Resources, Concurrency and Local Reasoning. In *CONCUR*, volume 3170 of *LNCS*, pages 49–67, 2004.
47. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local Reasoning about Programs that Alter Data Structures. In *CSL*, volume 2142 of *LNCS*, pages 1–19, 2001.
48. T. Ramananandro, G. Dos Reis, and X. Leroy. Formal verification of object layout for C++ multiple inheritance. In *POPL*, pages 67–80, 2011.
49. J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *PLDI*, pages 335–346, 2012.
50. V. Robert and X. Leroy. A Formally-Verified Alias Analysis. In *CPP*, volume 7679 of *LNCS*, pages 11–26, 2012.
51. J. G. Rossie and D. P. Friedman. An Algebraic Semantics of Subobjects. In *OOPSLA*, pages 187–199, 1995.
52. J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *JACM*, 60(3):22, 2013.
53. P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *CACM*, 53(7):89–97, 2010.
54. M. Sozeau. A New Look at Generalized Rewriting in Type Theory. *JFR*, 2(1), 2010.
55. B. Spitters and E. van der Weegen. Type Classes for Mathematics in Type Theory. *Mathematical Structures in Computer Science*, 21(4):795–825, 2011.
56. H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In *POPL*, pages 97–108, 2007.
57. V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Z. Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *POPL*, pages 209–220, 2015.